

Seria *Web* este coordonată de Sabin Buraga
(Facultatea de Informatică, Universitatea „A.I. Cuza”, Iași)

Ștefan Tanasă este cadru didactic la Facultatea de Informatică, Universitatea „A.I. Cuza”, Iași, unde predă cursul de *Proiectarea siturilor Web* și laboratoare de *Tehnologii Web, Sisteme de operare și Rețele de calculatoare*. Din noiembrie 2002, este înscris la doctorat cu tema *Medii virtuale distribuite*, în cadrul Universității „A.I. Cuza”. Este o continuare a activității sale de cercetare, care a debutat cu un grant finanțat de *Agenția Națională pentru Știință, Tehnică și Inovație*. Este permanent interesat de tehnologiile Web. Din 2001, este coordonator al WebGroup (grup de interes în domeniul tehnologiilor Web din cadrul Facultății de Informatică).

Cristian Olaru este analist programator la Direcția Județeană a Arhivelor Naționale Iași (DJAN) și a participat la dezvoltarea aplicației JAIS, destinată evidenței informatizate a fondurilor și colecțiilor DJAN Iași, implementată utilizând limbajul de programare **Java**, în special API-urile SWING, JDBC2, JAXP, JHelp, Java Printing, Java2D. De asemenea, se ocupă de pagina Web a acestei instituții. Este membru activ al WebGroup.

Ștefan Andrei este cadru didactic la Facultatea de Informatică, Universitatea „A.I. Cuza”, Iași, unde a predat și predă *cursuri de programare* (limbajele C, C++, Java și Prolog), *Construcția compilatoarelor*, *Limbaje formale*, *Navigare Internet* și *Sisteme distribuite*. A obținut doctoratul în domeniul informaticii cu teza *Procesare paralelă* susținută la Universitatea din Hamburg (Germania), este preocupat de procesarea datelor în diverse limbaje de programare. A acumulat o vastă experiență în studiile efectuate în străinătate și este permanent interesat de modelarea sistemelor distribuite și de noile tehnologii oferite de limbajul Java. De-a lungul timpului, a îndrumat mulți studenți pentru pregătirea lucrărilor de licență, majoritatea fiind implementate în limbajul Java.

© 2003 by Editura POLIROM

www.polirom.ro

Editura POLIROM

Iași, B-dul Copou nr. 4, P.O. BOX 266, 6600

București, B-dul I.C. Brătianu nr. 6, et. 7, ap. 33, O.P. 37; P.O. BOX 1-728, 70700

Descrierea CIP a Bibliotecii Naționale a României:

TANASĂ, ȘTEFAN

Java de la 0 la expert / Ștefan Tanasă, Cristian Olaru, Ștefan Andrei – Iași: Polirom, 2003

840 p.; 24 cm

ISBN: 973-681-201-4

I. Olaru, Cristian

II. Andrei, Ștefan

004.43 JAVA

Printed in ROMANIA

ȘTEFAN TANASĂ, CRISTIAN OLARU, ȘTEFAN ANDREI

Java

de la 0 la expert



130334
B.C.U. - IASI

POLIROM
2003

Tablă de materii

Cuvânt înainte	15
Prefață	17
1. INTRODUCERE	19
1.1. Cuvinte cheie	19
1.2. Istoricul și caracteristicile limbajului Java	20
1.3. Instalare	21
1.4. Tipuri de aplicații Java	23
1.5. Mașina virtuală Java	29
1.6. Concluzii	31
1.7. Test grilă	32
1.8. Exerciții propuse spre implementare	32
2. FUNDAMENTELE LIMBAJULUI JAVA	33
2.1. Cuvinte cheie	33
2.2. Fișiere sursă	34
2.3. Atomi lexicali	35
2.3.1. Caractere Unicode	36
2.3.2. Traduceri lexicale	37
2.4. Tipuri de date	39
2.5. Expresii și operatori	46
2.6. Variabile	64
2.7. Declarații și inițializări	69
2.8. Conversii	71
2.8.1. Contextele de conversie	76
2.9. Structuri de control	81
2.9.1. Instrucțiunea vidă	82
2.9.2. Instrucțiunea etichetă	82
2.9.3. Instrucțiunea expresie	83
2.9.4. Instrucțiunea if	84
2.9.5. Instrucțiunea switch	85
2.9.5. Instrucțiunea while	87
2.9.6. Instrucțiunea do	89
2.9.7. Instrucțiunea for	91
2.9.8. Instrucțiunea break	94

2.9.9. Instrucțiunea continue	95
2.9.10. Instrucțiunea return	96
2.9.11. Instrucțiunea throw	97
2.9.12. Instrucțiunea synchronized	99
2.9.13. Instrucțiunea try-catch-finally	100
2.10. Concluzii	105
2.11. Test grilă	107
2.12. Exerciții propuse spre implementare	113
2.13. Proiecte propuse spre implementare	114
3. CLASE, INTERFEȚE ȘI TABLOURI	117
3.1. Cuvinte cheie	117
3.2. Clase	118
3.2.1. Domeniul de vizibilitate al numelui unei clase	119
3.2.2. Modificatorii unei clase	121
3.2.3. Clase derivate	123
3.2.4. Clasa Object	125
3.2.5. Implementarea interfețelor	128
3.2.6. Declarațiile membrilor unei clase	130
3.2.6.1. Niveluri de acces	133
3.2.6.2. Declarațiile atributelor unei clase	138
3.2.6.3. Declarațiile metodelor unei clase	149
3.2.7. Inițializatori statici	160
3.2.8. Declarațiile constructorilor	161
3.2.9. Clase interioare	167
3.2.10. Distrugerea obiectelor și eliberarea memoriei	171
3.2.11. Împărțirea unei aplicații în fișiere	175
3.3. Interfețe	178
3.3.1. Declarațiile atributelor unei interfețe	179
3.3.2. Declarațiile metodelor unei interfețe	181
3.3.3. Moștenire multiplă prin intermediul interfețelor	184
3.4. Tablouri	188
3.5. Conversii ale tipului referință	196
3.5.1. Conversii implicite ale tipului referință	196
3.5.2. Conversii explicite ale tipului referință	198
3.6. Concluzii	204
3.7. Test grilă	205
3.8. Exerciții propuse spre implementare	225
4. ȘIRURI ȘI STRUCTURI DINAMICE DE DATE	229
4.1. Cuvinte cheie	229
4.2. Clasa String	230
4.3. Clasa StringBuffer (java.util.StringBuffer)	236
4.4. Clasa StringTokenizer (java.util.StringTokenizer)	239
4.5. Structuri dinamice de date	240
4.5.1. Liste simplu înlănțuite	240

4.5.2. Arbori binari	244
4.5.3. Enumeratori	249
4.5.3.1. Colecția Vector	250
4.5.4. Tabele de dispersie (hash)	254
4.5.5. Clasa Stack (java.util.Stack)	256
4.5.6. Clasa BitSet	257
4.5.7. Clasa Properties	258
4.6. Concluzii	263
4.7. Test grilă	263
4.8. Exerciții propuse spre implementare	264
5. FIȘIERE, FLUXURI DE DATE ȘI SERIALIZAREA OBIECTELOR	267
5.1. Cuvinte cheie	267
5.2. Introducere	268
5.3. Clase Java pentru intrări și ieșiri	271
5.3.1. Clasa File	271
5.3.2. Clasa RandomAccessFile	273
5.3.3. Clase Java pentru fluxuri de intrare/ieșire	277
5.3.3.1. Fluxuri de nivel jos	278
5.3.3.2. Fluxuri filtru de nivel înalt	281
5.3.3.3. Fluxuri pentru citire și fluxuri pentru scriere	283
5.3.3.4. Clasele PrintWriter și BufferedReader	286
5.3.3.5. Clasa StreamTokenizer	291
5.3.3.6. Clasa System	297
5.4. Serializarea obiectelor	301
5.4.1. Clasa ObjectOutputStream	302
5.4.2. Clasa ObjectInputStream	304
5.4.3. Interfața Serializable	307
5.4.4. Interfața Externalizable	308
5.4.5. Crearea unei clase serializabile	308
5.4.5.1. Derivarea unei clase serializabile	309
5.4.5.2. Implementarea interfeței Serializable	309
5.4.5.3. Metode de serializare obișnuite	311
5.4.5.4. Implementarea interfeței Externalizable	314
5.4.5.5. Declararea interfeței și a tuturor metodelor de acces	316
5.5. Concluzii	321
5.6. Test grilă	321
5.7. Exerciții propuse spre implementare	325
5.8. Proiecte propuse spre implementare	326
6. FIRE DE EXECUȚIE	327
6.1. Cuvinte cheie	327
6.2. Introducere	328
6.2.1. Breviar teoretic	328
6.2.2. Corectitudinea programelor	328
6.2.3. Primitivile de programare concurentă	329

6.3. Programare concurentă în Java	330
6.3.1. Crearea unui fir de execuție prin extinderea clasei Thread	330
6.3.2. Crearea unui thread utilizând interfața Runnable	332
6.3.3. Controlul unui fir de execuție	334
6.3.4. Prioritatea firelor de execuție	337
6.3.5. Metoda sleep	341
6.3.6. Starea de blocare	342
6.3.7. Grupuri de thread-uri	342
6.3.8. Stări monitor	345
6.3.9. Excludere mutuală și sincronizarea	346
6.3.10. Lucrul cu wait() și notify()	348
6.4. Studii de caz: problema filosofilor și problema producător-consumator	357
6.4.1. Problema filosofilor	357
6.4.2. Problema producător-consumator	361
6.5. Concluzii	367
6.6. Test grilă	367
6.7. Exerciții propuse spre implementare	368
6.8. Proiecte propuse spre implementare	368
7. APPLETURI ȘI INSTRUMENTE DE LUCRU CU FERESTRELE (AWT)	369
7.1. Cuvinte cheie	369
7.2. Appleturi	370
7.2.1. Clasa Applet (java.applet.Applet)	370
7.2.2. Obținerea parametrilor de intrare a appleturilor	373
7.2.3. Contextul unui applet	375
7.2.4. Comunicarea dintre appleturi	376
7.2.5. Redarea sunetelor și imaginilor	378
7.2.5.1. Afișarea conținutului grafic	378
7.2.5.2. Redarea sunetelor	380
7.2.6. Transformarea unui applet într-o aplicație de sine stătătoare	381
7.3. Grafica în Java	383
7.4. Componente și evenimente	387
7.4.1. Bucla evenimentelor	388
7.4.2. Componente	392
7.4.3. Containere	393
7.4.4. Etichete	394
7.4.5. Butoane	395
7.4.6. Câmpuri și arii text	397
7.4.6.1. Clasa java.awt.TextComponent	397
7.4.6.2. Clasa java.awt.TextField	398
7.4.6.3. Clasa java.awt.TextArea	400
7.4.7. Butoane de selecție (check-box și radio)	405
7.4.8. Liste de opțiuni	407
7.4.9. Suprafețe de desenare (cavasses)	411
7.4.10. Panouri (panels)	412
7.4.11. Meniuri Popup	413

7.5. Gestionări de poziționare (Layout Managers)	416
7.5.1. Gestionarul BorderLayout	416
7.5.2. Gestionarul CardLayout	419
7.5.3. Gestionarul FlowLayout	421
7.5.4. Gestionarul GridLayout	422
7.5.5. Gestionarul GridBagLayout	424
7.5.6. Poziționarea absolută	426
7.6. Ferestre	427
7.6.1. Clasa java.awt.Window	427
7.6.2. Clasa java.awt.Frame	429
7.6.3. Clasa java.awt.Dialog	430
7.6.4. Clasa java.awt.FileDialog	432
7.7. Managerul de securitate	434
7.8. Concluzii	436
7.9. Test grilă	437
7.10. Exerciții propuse spre implementare	439
7.11. Proiecte propuse spre implementare	440
8. ACCESUL LA BAZE DE DATE FOLOSIND JDBC	441
8.1. Cuvinte cheie	441
8.2. Introducere	442
8.3. Clasificarea driverelor JDBC	444
8.4. Breviar SQL	445
8.4.1. Instrucțiunea CREATE TABLE	446
8.4.2. Instrucțiunea INSERT	446
8.4.3. Instrucțiunea UPDATE	447
8.4.4. Instrucțiunea DELETE	447
8.4.5. Instrucțiunea SELECT	448
8.5. Mapări de tipuri între SQL și Java	448
8.6. Accesarea unei baze de date folosind JDBC	451
8.6.1. Înregistrarea driverului JDBC folosind clasa DriverManager	451
8.6.2. Stabilirea unei conexiuni către baza de date	452
8.6.3. Execuția unei instrucțiuni SQL	453
8.6.4. Procesarea rezultatelor	456
8.6.5. Închiderea unei conexiuni la o bază de date	460
8.7. Instalarea și configurarea MySQL	461
8.7.1. Introducere	461
8.7.2. Driver JDBC pentru MySQL	462
8.8. Utilizarea punții JDBC-ODBC	465
8.9. Tranzacții	468
8.9.1. Motivații pentru folosirea tranzacțiilor	468
8.9.2. ACID	470
8.9.3. Niveluri de izolare	471
8.10. Concluzii	472
8.11. Test grilă	472

8.12. Exerciții propuse spre implementare	474
8.13. Proiecte propuse spre implementare	475
9. PROGRAMAREA REȚELOR	477
9.1. Cuvinte cheie	477
9.2. Introducere	478
9.2.1. Adrese, porturi și socketuri	478
9.3. Programarea rețelor prin intermediul conexiunilor	481
9.3.1. Clasa ServerSocket	481
9.3.2. Clasa Socket	482
9.3.3. O aplicație simplă client/server orientată conexiune	484
9.4. Programarea rețelor prin intermediul datagramelor	489
9.4.1. Clasa DatagramPacket	489
9.4.2. Clasa DatagramSocket	491
9.4.3. O aplicație simplă client/server neorientată conexiune	493
9.4.4. Clasa InetAddress	496
9.4.5. Clasa URL	498
9.4.6. Obținerea unei pagini Web prin intermediul unui socket	505
9.5. Apelul metodelor la distanță	508
9.5.1. Obiectul de la distanță	510
9.5.2. Registrul de nume	511
9.5.3. Exemplu de utilizare a apelurilor la distanță	512
9.5.4. Localizarea fișierelor RMI. Pachete necesare	514
9.5.5. Clasa Naming	515
9.5.6. Clasa LocateRegistry	516
9.5.7. Interfața Registry	516
9.5.8. Clasa RemoteObject	517
9.5.9. Obiecte la distanță ca parametri	517
9.5.10. Serializarea unui obiect la distanță	519
9.5.11. Clasa RemoteServer	519
9.5.12. Clasa RemoteStub	520
9.5.13. Interfața Unreferenced	520
9.5.14. Clasa RMISocketFactory	520
9.5.15. Interfața RMIFailureHandler	522
9.5.16. Exemplu de aplicație RMI	522
9.6. Concluzii	527
9.7. Test grilă	528
9.8. Exerciții propuse spre implementare	529
9.9. Proiecte propuse spre implementare	529
10. SERVLETURI	531
10.1. Cuvinte cheie	531
10.2. CGI (Common Gateway Interface)	532
10.2.1. Variabilele de mediu	533
10.2.2. Apelarea programelor CGI din formularele HTML	535
10.2.3. Procesarea datelor din formularele HTML prin metoda GET	536

10.2.4. Procesarea unui formular prin metoda POST	538
10.3. Servleturi	540
10.3.1. Servlet API	541
10.3.2. Funcționalitatea servleturilor	541
10.3.3. Execuția servleturilor	542
10.3.4. Structura de bază a unui servlet	543
10.3.5. Ciclul de viață al unui servlet	544
10.3.6. Clasa HttpServlet	546
10.3.7. Redirecționarea cererii	552
10.3.8. Returnarea unei erori	553
10.3.9. Exemplu de aplicație Web	554
10.4. Concluzii	565
10.5. Test grilă	566
10.6. Exerciții propuse spre implementare	567
10.7. Proiecte propuse spre implementare	567
11. JAVA SERVER PAGES (JSP)	569
11.1. Cuvinte cheie	569
11.2. Introducere	570
11.3. Elemente JSP	571
11.3.1. Crearea paginilor JSP	571
11.3.2. Comentarii	572
11.3.3. Directive	574
11.3.3.1. Directiva page	574
11.3.3.2. Directiva include	575
11.3.3.3. Directiva taglib	576
11.3.4. Declarații	576
11.3.5. Inițializarea și terminarea unui JSP	577
11.3.6. Obiecte implicite	578
11.3.7. Expresii	579
11.3.8. Scriptlet-uri	579
11.3.9. Acțiuni	580
11.3.9.1. Integrarea componentelor <i>JavaBean</i>	581
11.4. Concluzii	582
11.5. Test grilă	583
11.6. Exerciții propuse spre implementare	583
11.7. Proiecte propuse spre implementare	583
12. PROCESAREA DOCUMENTELOR XML	585
12.1. Cuvinte cheie	585
12.2. Standarde și specificații	586
12.3. Instalare	587
12.4. SAX	587
12.4.1. Pachete necesare	587
12.4.2. Exemplu de document XML	588
12.4.3. Obținerea informațiilor din documentele XML	589

12.4.4. Obținerea valorilor atributelor unui tag	592	13.5.4. JSplitPane	687
12.4.5. Procesarea documentelor XML care conțin spații de nume	593	13.5.5. JTabbedPane	689
12.4.6. Procesarea documentelor XML care conțin DTD	596	13.6. Componente atomice simple	694
12.5. DOM	599	13.6.1. JLabel	694
12.5.1. Pachete necesare	600	13.6.2. Butoane	696
12.5.2. Crearea arborelui asociat unui document XML	600	13.6.2.1. JButton	697
12.5.3. Manipularea arborelui DOM	601	13.6.2.2. JToggleButton, JCheckBox, JRadioButton	698
12.5.3.1. Interfața Node	601	13.6.3. Borduri	700
12.5.3.2. Procesarea documentelor XML care utilizează spații de nume	608	13.6.4. JList	704
12.5.3.3. Procesarea documentelor XML care conțin DTD	611	13.6.5. JComboBox	708
12.5.3.4. Interfața Document	613	13.6.6. JSpinner	711
12.6. XSLT	616	13.7. Componente atomice complexe	712
12.6.1. Pachete necesare	617	13.7.1. Componente text	712
12.6.2. Realizarea unei transformări	617	13.7.2. JTable	722
12.7. Concluzii	621	13.7.3. JTree	734
12.8. Test grilă	622	13.8. Meniuri și bare de unelte	740
12.9. Exerciții propuse	622	13.8.1. Meniuri	740
12.10. Proiecte propuse	623	13.8.2. JToolBar	743
13. SWING	625	13.9. Dialoguri	748
13.1. Cuvinte cheie	625	13.9.1. JOptionPane	748
13.2. Introducere	626	13.9.2. JFileChooser	753
13.2.1. Obiective urmărite în acest capitol	626	13.9.3. JColorChooser	758
13.2.2. Despre interfețe grafice	627	13.10. Componente pentru progres și derulare	762
13.2.3. JFC (Java Foundation Classes)	628	13.10.1. JSlider	763
13.2.4. Componentele și pachetele librăriei Swing	630	13.10.2. JScrollBar	764
13.2.5. Principii de bază	636	13.10.3. JProgressBar	767
13.2.5.1. MVC (Model View Controller)	636	13.10.4. JToolTip	769
13.2.5.2. UI Delegate	637	13.11. Ferestre interioare	770
13.2.5.3. Look-and-feel	638	13.11.1. JDesktopPane	770
13.2.5.4. Mai multe despre modele	641	13.11.2. JInternalFrame	771
13.2.5.5. Swing versus AWT	644	13.12. Recomandări și optimizări	776
13.3. Fundamentele Swing	644	13.13. Concluzii	777
13.3.1. Evenimente și ascultători	644	13.14. Test grilă	778
13.3.2. Fire de execuție în Swing	656	13.15. Exerciții propuse spre implementare	783
13.3.3. Desenarea componentelor grafice	659	13.16. Proiecte propuse spre implementare	784
13.3.4. Alinierea la JavaBeans	664	14. SISTEMUL HELP PENTRU JAVA	785
13.3.5. Gestiunea „focusului”	666	14.1. Cuvinte cheie	785
13.3.5. Clasificarea componentelor Swing	669	14.2. Convenții	786
13.4. Conținere de bază	670	14.3. Principii de bază	786
13.4.1. JFrame	671	14.3.1. Introducere	786
13.4.2. Clase și interfețe legate de ferestre	676	14.3.2. Prezentarea pachetului JavaHelp	787
13.4.3. JWindow	677	14.3.3. Alcătuirea ferestrei standard JavaHelp	788
13.4.4. JDialog	679	14.3.4. Principii pe care sistemul JavaHelp le respectă	789
13.4.5. JApplet	681	14.4. Dezvoltarea efectivă a help-ului	789
13.5. Container intermediare	682	14.4.1. Temele HTML	790
13.5.2. JPanel	682	14.4.2. Fișierul HelpSet	791
13.5.3. JScrollPane	683		

14.4.3. Fișierul Map	793
14.4.4. Fișierul cuprins	795
14.4.5. Fișierul index	796
14.4.6. Indexarea și căutarea	798
14.4.7. Compresia și încapsularea	800
14.5. Atașarea de help aplicațiilor	800
14.5.1. Clasa HelpSet	801
14.5.2. Clasa HelpBroker	802
14.5.3. Clasa CSH	804
14.5.4. Adăugare de help ferestrelor	805
14.5.5. Adăugarea help-ului componentelor	805
14.5.6. Activarea help-ului prin apăsarea butoanelor	806
14.5.7. Help activat folosind mouse-ul	807
14.5.8. Scufundarea help-ului în aplicație	808
14.6. Concluzii	809
14.7. Test grilă	809
14.8. Exerciții propuse spre implementare	810
14.9. Proiecte propuse spre implementare	810
15. INTERNAȚIONALIZAREA APLICAȚIILOR	813
15.1. Cuvinte cheie	813
15.2. Introducere	814
15.3. Codări	815
15.3.1. Despre Unicode	816
15.3.2. Seturi de caractere	817
15.4. Setarea localizării	819
15.5. Separarea datelor	822
15.6. Formatări	826
15.6.1. Formatarea numerelor	826
15.6.2. Reprezentarea monetară	826
15.6.3. Formatarea datei calendaristice și a orei	827
15.6.4. Formatarea mesajelor	828
15.7. Lucrul cu fonturi	829
15.8. Introducerea textului	830
15.9. Pașii necesari pentru internaționalizarea aplicațiilor	831
15.10. Concluzii	832
15.11. Test grilă	832
15.12. Exerciții propuse spre implementare	834
15.13. Proiecte propuse spre implementare	834
Bibliografie	837

Cuvânt înainte

Java este un limbaj de programare orientat obiect consacrat. Cele mai multe aplicații distribuite sunt scrise în Java, iar noile evoluții tehnologice permit utilizarea sa și pe dispozitive mobile gen telefon, agendă electronică, palmtop etc. În felul acesta se creează o platformă unică, la nivelul programatorului, deasupra unui mediu eterogen extrem de diversificat. Avantajele sunt evidente, atât pentru proiectanți, care „scriu o dată și execută pe orice mașină virtuală Java (JVM)”, cât și pentru utilizatori, care vor beneficia de un spectru îmbogățit de servicii. De aceea se poate afirma cu încredere că Java este un câștigător, în lumea volatilă a tehnologiilor de calcul.

Cartea de față este un instrument foarte util pentru cei care doresc să învețe Java și să scrie aplicații distribuite sau, cu alt termen, aplicații în rețea. Cu entuziasm și dedicare, cei trei autori au cuprins în opera lor, într-un stil facil și plăcut, aspectele caracteristice ale limbajului Java, cât și ale tehnologiilor care îl însoțesc și permit scrierea de aplicații distribuite. Fiecare capitol are multe exemple de programare, bine documentate, și se încheie cu întrebări, exerciții și proiecte propuse, care au rolul de a fixa cunoștințele asimilate. În felul acesta, învățarea limbajului de programare Java devine un exercițiu practic, reconfortant.

Remarc cu plăcere că activitatea grupului de tineri de la Facultatea de Informatică din Iași, care studiază tehnologiile Web și de calcul distribuit, se îmbogățește atât calitativ, cât și ca producție de carte de specialitate. Este un aspect pozitiv, cu influențe importante asupra procesului didactic.

Prof. Dr. Ing. Dan Grigoraș
Computer Science Department
University College, Cork, Ireland
Facultatea de Informatică
Universitatea „Al.I. Cuza”, Iași, România
6 februarie 2003

Prefață

Java este un limbaj de programare de nivel înalt, orientat obiect, proiectat inițial pentru realizarea de aplicații pentru Internet și mai cu seamă pentru Web. Acesta este utilizat în prezent cu succes și pentru programarea aplicațiilor destinate intranet-urilor. În acest sens, multe firme recurg la limbajul Java în procesul de informatizare întrucât oferă un foarte bun suport pentru tehnologiile de vârf și, nu în ultimul rând, pentru faptul că este gratuit și în mod continuu îmbogățit și îmbunătățit.

Cititorului i se oferă un ghid complet, cartea de față cuprinzând atât noțiunile de bază ale limbajului Java, cât și elemente de nivel mediu și avansat. De asemenea, sunt prezentate noțiuni de programare orientată obiect și algoritmică, elemente de programare concurrentă și distribuită, toate acestea pentru ca limbajul Java să fie mai ușor de înțeles. De aceea, cartea poate fi un instrument util pentru însușirea și consolidarea cunoștințelor pentru orice elev, student, programator și nu numai.

Primele patru capitole cuprind noțiuni de bază, de la instalare, compilare și execuție, la tipuri de date și instrucțiuni. Aceste prime capitole au rolul de a iniția cititorul în limbajul de programare Java aducându-l la un nivel mediu. Celelalte capitole prezintă treptat tehnologiile Java utile în realizarea de către programatori a aplicațiilor, de la nivel mediu până la expert.

Cartea este bogată în exemple de programe Java scurte și comentate pentru a ilustra cât mai bine cele prezentate. De regulă, fiecare capitol se termină cu prezentarea unei aplicații concrete pentru a realiza trecerea cititorului de la un cunoscător al limbajului Java la un programator Java. Toate capitolele au o structură de prezentare asemănătoare, începând cu secțiunea de cuvinte cheie, apoi conținutul efectiv, urmate de concluzii, test grilă, exerciții și proiecte propuse spre implementare. Secțiunea de test grilă conferă cititorului suportul pentru obținerea de certificate internaționale Java (de exemplu, <http://suned.sun.com/US/certification>, <http://studyguides.cramsession.com/cramsession/sun/java2>, <http://enterprisedeveloper.com/certification>).

Pentru a putea stabili o legătură cu cei interesați de limbajul Java și pentru a furniza informații suplimentare, cartea va avea la adresa <http://www.infoiasi.ro/~stanasa/java> o pagină de prezentare, conținând rezolvarea testelor grilă și eventual a exercițiilor propuse, alte exemple de programe Java, prezentarea unor unelte de programare Java etc.

Pentru suportul acordat realizării acestei cărți mulțumim domnilor Sabin-Corneliu Burăga, Cristian Frăsinaru, Nicolae Tăbușcă, Laurențiu Constantin și Bogdan Danu.

De asemenea, mulțumim familiilor noastre pentru tot sprijinul acordat.

Autorii
februarie 2003

1. Introducere

1.1. Cuvinte cheie

- limbaj orientat obiect, compilator, interpretor, platformă
- instalare, Windows, Linux
- Internet, Web, HTML, applet, servlet
- Java, C, C++, Pascal, Visual Basic

1.2. Istoricul și caracteristicile limbajului Java

Java este un limbaj de programare orientat obiect, destinat în principal programării Internet.

Începutul limbajului Java este în toamna anului 1991, când firma Sun Microsystems a finanțat un proiect cu numele *Green* condus de James Gosling. Scopul echipei *Green* era să plaseze firma Sun Microsystems pe piața produselor electronice comerciale. Inginerii și dezvoltatorii de soft de la Sun au căutat microprocesoare care să ruleze pe o multitudine de mașini, în particular pe sisteme distribuite care lucrează în timp real. Cheia succesului firmei Sun a fost abilitatea lucrului pe platforme multiple. După patru ani de lucru, echipa *Green* finalizează specificațiile limbajului Java. Apoi, compania Sun Microsystems vinde licența firmelor IBM, Microsoft, Silicon Graphics, Adobe și Netscape.

Caracteristicile limbajului Java sunt:

- limbaj *interpretat și compilat*. Un limbaj este *interpretat* dacă instrucțiunile unui program scris în acel limbaj sunt procesate linie cu linie și traduse în cod mașină. Un limbaj este *compilat* dacă un program scris în acel limbaj este tradus într-un cod pe care calculatorul îl poate „înțelege” (executa) mult mai ușor. Programele interpretate sunt mai lente decât cele compilate, însă cele compilate sunt de obicei dependente de platforma respectivă. Programele Java sunt mai întâi compilate în niște fișiere intermediare asemănătoare codului de asamblare (numite *byte code*, eng.), apoi acestea sunt interpretate de mediul de execuție Java în instrucțiuni mașină asociate platformei sistem.
- limbaj *independent de platformă*. La instalarea limbajului Java, se va crea o mașină virtuală Java care are drept scop traducerea instrucțiunilor unui *byte code* Java în instrucțiuni mașină pentru platforma curentă. Astfel fișierele intermediare *byte code* pot fi copiate și executate pe orice platformă (indiferent că este Windows, Unix, Solaris etc.).
- limbaj *orientat obiect*. Cea mai importantă proprietate a limbajului Java este orientarea obiect, aceasta fiind privită ca o trăsătură implicită. Java pune în evidență toate aspectele legate de programarea orientată obiect: obiecte, trimitere de parametri, încapsulare, clase, biblioteci, moștenire și modificatori de acces.
- limbaj *concurrent*. Concurența (eng. *multithreading*) înseamnă capacitatea unui program de a executa mai multe secvențe de cod în același timp. O secvență de cod Java se numește *fir de execuție* (eng. *thread*). Datorită posibilității creării mai multor fire de execuție, un program Java poate să execute mai multe sarcini simultan, de exemplu animația unei imagini, transmiterea unei melodii spre placa de sunet, comunicarea cu un server, alocarea și eliberarea memoriei etc.).
- limbaj *simplu*. Spre deosebire de C++, care este cel mai popular limbaj orientat obiect, Java elimină câteva dintre trăsăturile acestuia: posibilitatea moștenirii multiple este exclusă, șirurile sunt încapsulate într-o structură clasă. Spre deosebire de C/C++, în Java nu există pointeri, iar alocarea și dealocarea

memoriei se fac automat. Există un mecanism de eliberare a memoriei (eng. *garbage collection*) pus în practică de un fir de execuție de prioritate mică.

- limbaj *distribuit*. Java este distribuit deoarece permite utilizarea obiectelor locale și de la distanță. Limbajul Java oferă posibilitatea dezvoltării de aplicații pentru Internet, capabile să ruleze pe platforme distribuite și eterogene. În acest sens, Java respectă standardul IEEE (eng. *Institute of Electrical and Electronics Engineers*) pentru structurile de date, cum ar fi folosirea întregilor, numerelor în virgulă flotantă și șirurilor de caractere. Java se poate utiliza în aplicații de rețea, deoarece respectă protocoalele de rețea, cum ar fi FTP, HTTP etc.
- limbaj *performant*. Interpretorul Java este capabil să execute un *byte code* aproape la fel de repede ca un cod compilat. Având posibilitatea să lucreze cu fire de execuție multiple, Java justifică faptul că este un limbaj performant. De exemplu, un program Java poate aștepta citirea unor date, în timp ce un alt fir de execuție poate alocă sau elibera memoria necesară programului.
- limbaj *dinamic și robust*. Java întârzie alocarea obiectelor și legarea dinamică a claselor pentru momentul execuției. Astfel, se vor evita erorile de alocare chiar dacă mediul s-a schimbat de la ultima compilare a programului. Faptul că Java este robust se justifică prin eliminarea utilizării pointerilor, care erau generatoare de erori în cazul programelor C/C++. În schimb, Java verifică memoria dinamic înainte de a fi alocată și are un sistem automat de alocare/dealocare a memoriei.
- limbaj *sigur*. Programele Java nu pot accesa memoria *heap*, *stack* sau alte secțiuni protejate de memorie, deoarece Java nu folosește pointeri și alocă memorie doar la execuție. Înainte ca interpretorul Java să execute *byte* codul, se verifică dacă este un cod Java valid prin cercetarea accesului la date, conversiilor de date nepermise, valori și parametri incorecți, depășirea stivei.

1.3. Instalare

Limbajul Java se poate instala pe calculatoare cu diverse sisteme de operare, cum ar fi: Windows (95, 98, NT, ME, 2000, XP), Linux, stații Sun Solaris. Acest lucru este posibil deoarece un program Java nu se execută direct de microprocesor, ci utilizând un calculator ipotetic intermediar numit *mașina virtuală Java* (eng. *Java Virtual Machine - JVM*). Un compilator Java translatează codul sursă Java într-un limbaj numit *byte code* (care sunt instrucțiuni mașină pentru mașina virtuală Java) asemănător limbajului de asamblare și pe care îl vom denumi în paginile acestei cărți *cod binar Java*. Pentru execuția unui program Java, interpretorul Java inspectează și decodifică codul binar Java, îl verifică dacă este sigur de executat și apoi îl execută instrucțiune cu instrucțiune. La instalarea limbajului Java, se realizează de fapt corespondența dintre acțiunile codului binar Java și instrucțiunile mașinii fizice. Acesta este „secretul” independenței de platformă a limbajului Java. Această caracteristică este moștenită și de C++.

Faptul că în procesul de traducere a codului sursă Java în instrucțiunile mașinii fizice s-a interpus mașina virtuală Java, implică o scădere a vitezei de execuție față de

un program echivalent care folosește instrucțiuni mașină native. Dacă la începuturile limbajului Java un program era de 10 ori mai lent, în versiunile actuale ale pachetului JDK (eng. *Java Development Kit*) performanțele au crescut semnificativ, raportul vitezei de execuție dintre un program Java și un program scris în cod nativ fiind de aproximativ 2. Mai mult, pentru un mediu Java care suportă *compilarea în timp* *încărcării* (eng. *Just-In-Time compilation*) nu va exista nici o diferență de viteză a execuției. Compilatoarele Just-In-Time vor traduce programele Java în instrucțiuni mașină native chiar din momentul încărcării.

Sunt disponibile mai multe medii de dezvoltare a aplicațiilor Java, realizate de diverși producători printre care Sun, Borland și Symantec. Firma Sun pune la dispoziție gratuit kitul de instalare pentru sistemele de operare Windows, Linux și Solaris pe siturile <http://java.sun.com/> și <http://www.javasoft.com>. O versiune a kitului de instalare pentru sistemul de operare MacOS este disponibilă pe situl <http://devworld.apple.com/java/>. Există editoare și medii de programare utile lucrului cu programe Java, printre care: editoare de texte pe situl <http://www.download.com/>, Kawa pe siturile <http://www.allaire.com/Products/index.cfm> și <http://www.macromedia.com/software/kawa>, JBuilder la adresa <http://www.borland.com/jbuilder>, JCreator la situl <http://www.JCreator.com/>.

Instrucțiuni detaliate despre modul de instalare a mediului Java se găsesc pe siturile care oferă kiturile de instalare.

Pentru sistemul de operare Windows, instalarea kitului JDK 1.4.1 se va face într-un director C:\j2sdk1.4.1\ care va conține subdirectoarele bin, demo, include, jre și lib. În cadrul subdirectorului bin se găsesc principalele instrumente ale pachetului JDK:

- javac.exe – compilatorul de Java;
- java.exe – interpretorul de Java;
- appletviewer.exe – instrument de vizualizat appleturi;
- javadoc.exe – generator de documentații;
- jar.exe – instrument pentru arhive jar;
- javap.exe – dezasamblor de fișiere byte code.

Apoi se poate instala și documentația, pentru care se va crea un subdirector docs. Directorul C:\j2sdk1.4.1\bin trebuie să fie prezent în variabila PATH din fișierul autoexec.bat. Codul sursă al bibliotecilor claselor este inclus în arhiva src.jar (în versiunile mai vechi) sau src.zip din directorul C:\j2sdk1.4.1\. Fișierele sursă se pot extrage folosind utilitarul jar astfel:

```
jar xvf src.jar
```

sau:

```
jar xvf src.zip
```

Așadar, se va crea un subdirector C:\j2sdk1.4.1\src care va conține fișiere cod-sursă Java dezarhivate. Dezarhivarea surselor nu este obligatorie.

Pentru sistemul Linux va trebui să procurăm un fișier cu extensia .bin, de exemplu pentru JDK 1.4.1 există j2sdk-1_4_1_01-linux-i586-rpm.bin. Acesta va fi executat de administratorul sistemului (utilizatorul root) astfel:

```
./j2sdk-1_4_1_01-linux-i586-rpm.bin
```

Vor fi afișați termenii de utilizare a mediului Java și, în cazul în care suntem de acord, vom tasta cuvântul yes. La execuția acestui program va rezulta fișierul j2sdk-1_4_1_01-fcs-linux-i586.rpm, cu care vom realiza instalarea efectivă:

```
rpm -i j2sdk-1_4_1_01-fcs-linux-i586.rpm
```

Pachetul JDK va fi instalat în directorul /usr/java/j2sdk1.4.1_01/, iar pentru a putea utiliza mai ușor comenzile puse la dispoziție vom edita fișierul .bash_profile din contul personal adăugând următoarea linie:

```
PATH=$PATH:$HOME/bin:/usr/java/j2sdk1.4.1_01/bin
```

Astfel, directorul care conține comenzile din mediul Java a fost inclus în calea de căutare a fișierelor executabile. Structura de directoare este similară cu cea din sistemul Windows, diferența fiind la numele comenzilor, acestea neavând extensie (în Windows acestea au extensia .exe). De asemenea, comenzile sunt utilizate cu aceiași parametri ca pentru sistemul Windows.

1.4. Tipuri de aplicații Java

În principiu, există mai multe tipuri de aplicații Java:

- a) aplicații de sine stătătoare (eng. *stand-alone*);
- b) aplicații care se execută pe partea de client (numite appleturi);
- c) aplicații care se execută pe partea de server (numite servleturi).

a) O aplicație de sine stătătoare Java este un program care poate fi executat în afara contextului unui browser (navigător) Web. Principala caracteristică a acestui tip de aplicație este încapsularea în cadrul clasei principale a unei funcții main(), asemănătoare cu cea folosită în C, care are următoarea semnătură:

```
public static void main(String [] args).
```

Aplicațiile Java de sine stătătoare se pot împărți la rândul lor în:

- aplicații de consolă (de exemplu, fereastră DOS pe un calculator personal rulând Windows);
- aplicații fereastră (care lucrează cu ferestre grafice folosind interfața grafică cu utilizatorul [eng. *Graphical User Interface - GUI*]).

În continuare vom edita un text sursă pe care îl vom salva în fișierul AplicatiaUnu.java ca aplicație de consolă.

```
public class AplicatiaUnu {
    public static void main (String args[]) {
        if (args.length == 0)
            System.out.println("Aceasta este o aplicatie independenta.");
    }
}
```

```

else {
    System.out.println("Argumentele programului sunt: ");
    for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);
}
}
}

```

Chiar dacă este un program simplu, acesta înglobează multe dintre trăsăturile unui program orientat obiect complex, și anume: are un obiect (implicit generat de sistemul Java), utilizează încapsularea și trimiterea parametrilor, definește o clasă, folosește moștenirea și modificatori de acces. Compilarea acestui program Java se realizează cu comanda:

```
javac AplicatiaUnu.java
```

Astfel, se va crea fișierul în cod binar Java cu numele `AplicatiaUnu.class`. Execuția acestui program se va realiza folosind comanda:

```
java AplicatiaUnu
```

Pe ecran va apărea mesajul:

```
Aceasta este o aplicatie independenta.
```

De remarcat faptul că la compilare trebuie scrisă și extensia fișierului (`.java`), pe când la interpretare nu trebuie specificată extensia.

Continuăm cu câteva explicații asupra codului Java de mai sus. Fișierul constă în definiția unei clase declarate public intitulată `AplicatiaUnu` (de altfel, aceasta este o cerință obligatorie: numele clasei declarate public coincide cu numele fișierului Java). Metoda `main()` este punctul normal de intrare pentru o aplicație Java.

Aceasta este declarată public prin convenție. Însă trebuie declarată obligatoriu static pentru a putea fi executată fără a construi o instanță a clasei corespunzătoare. Șirul `args[]` conține argumentele pe care utilizatorul poate să le introducă la linia de comandă. De exemplu, dacă executăm programul `AplicatiaUnu` astfel:

```
java AplicatiaUnu Aceasta este o aplicatie independenta
```

obținem `args[0]` egal cu „Aceasta” și `args[1]` egal cu „este”, și tot așa până la `args[4]`, care va avea „independenta”. Atenție, ca și în C, șirul începe de la locația de index 0 și se termină la locația de index 4 (lungimea șirului `args` fiind 5). Numele clasei (`AplicatiaUnu`) și numele comenzii (`java`) nu apar în șirul `args` (cum se întâmplă în limbajul C, de exemplu). Numele dat acestui argument (`args`) este orientativ și poate fi schimbat după dorință.

Clasa `System` conține câmpul:

```
public static final PrintStream out;
```

care reprezintă fluxul de ieșire standard. Acest flux este deja deschis și pregătit să accepte afișarea datelor.

În clasa `PrintStream` există metoda:

```
public void println(String x);
```

care are drept scop afișarea șirului `x` și terminarea liniei. Astfel, linia:

```
System.out.println("Aceasta este o aplicatie independenta.");
```

va avea drept scop afișarea la consolă a mesajului „Aceasta este o aplicatie independenta”.

b) Aplicațiile care se execută pe partea de client sunt cele încărcate de pe un server (de cele mai multe ori aflat la distanță) și apoi executate de programe speciale cum ar fi navigatoarele Web. De exemplu, appleturile sunt aplicații care sunt executate de către navigatorul Web, iar middleturile sunt executate de către dispozitive mobile fără fir (mai exact de către sistemul software al acestora).

Un applet este un program Java care respectă o mulțime de convenții care îi permit să ruleze în cadrul unui navigator Web ce încorporează o mașină virtuală Java. Tehnologia appleturilor este similară, într-un anumit sens, cu a programelor JavaScript. Fișierul cu extensia `.class` al unui applet este stocat pe un server Web și poate fi accesat de către un client prin intermediul unei pagini care conține acel applet. La încărcarea paginii, codul appletului este transferat pe mașina client și va fi executat cu ajutorul mașinii virtuale incluse în navigator.

Atenție! Navigatorul Web trebuie să aibă suport pentru rularea appleturilor!

Fie textul sursă Java memorat în fișierul `AplicatiaDoi.java`:

```

import java.awt.Graphics;
import java.applet.Applet;

public class AplicatiaDoi extends Applet {
    public void paint(Graphics g) {
        g.drawString("Acesta este un applet", 50, 50);
    }
}

```

Compilarea acestui program Java se realizează astfel:

```
javac AplicatiaDoi.java
```

Astfel se va crea fișierul *byte code* cu numele `AplicatiaDoi.class`. Apoi vom scrie un fișier HTML, intitulat `AplicatiaDoi.html`, care va avea inclus marcator `<applet>`. De exemplu, ar putea fi:

```

<html>
<head>
    <title>Exemplu de applet</title>
</head>

```

```
<body>
  <applet code="AplicatiaDoi.class" width="300" height="200">
  </applet>
</body>
</html>
```

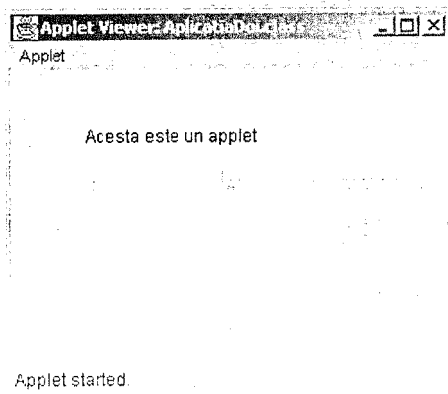
În versiunea HTML 4.0 marcatorul `<applet>` este considerat învechit (eng. *deprecate*). Ca alternativă se folosește marcatorul `<object>`, însă nu toate navigatoarele oferă suport pentru appleturi prin intermediul acestui tag. De exemplu:

```
<html>
<head>
  <title>Exemplu de applet folosind eticheta obiect</title>
</head>

<body>
  <object codetype="application/java"
    classid="java:AplicatiaDoi.class"
    width="300" height="200">
  </object>
</body>
</html>
```

Execuția acestui program se va realiza prin intermediul unui navigator Web (de exemplu, Netscape Navigator sau Microsoft Internet Explorer) care va apela fișierul `AplicatiaDoi.html` sau rulând un vizualizator de appleturi cu comanda:

```
appletviewer AplicatiaDoi.html
```



Continuăm cu câteva explicații asupra codului Java de mai sus. Fișierul constă în definiția unei clase declarate `public` intitulată `AplicatiaDoi`. Mai întâi, avem

două instrucțiuni de import al unor clase. Este vorba de clasa `Graphics` din pachetul `awt` (Abstract Windows Toolkit) și clasa `Applet` din pachetul `java.applet`. Urmează apoi declarația clasei noastre intitulată `AplicatiaDoi`. Prezența cuvântului rezervat `extends` semnifică derivarea clasei `AplicatiaDoi` din clasa `Applet`. Urmează definiția metodei `paint()` care aparține clasei `Component`. Clasa `Component` este situată trei niveluri mai sus decât clasa `Applet`, adică ierarhia claselor este: `Object`, `Component`, `Container`, `Panel`, `Applet`. Metoda `paint()` este automat apelată când appletul decide că este nevoie de actualizarea sa, de exemplu când appletul este plasat prima dată pe ecran, când s-a mutat appletul sau când altă fereastră a fost deasupra appletului. De fapt, appletul apelează metoda `update()` din aceeași clasă `Component`, care la rândul ei apelează metoda `paint()`. Sintaxa metodei `paint()` este:

```
public void paint(Graphics g);
```

Obiectul `g` din clasa `Graphics` reprezintă suprafața pe care se poate desena, în cazul unui applet fiind aria din interiorul appletului. Clasa `Graphics` are foarte multe metode, însă cea mai folosită metodă pentru afișarea unui șir de caractere este `drawString()`. Această metodă are sintaxa:

```
public abstract void drawString(String str, int x, int y);
```

Apelul acestei metode asupra unui obiect din clasa `Graphics` are ca efect afișarea șirului `str` începând de la poziția `(x, y)` a appletului. Menționăm că colțul din stânga sus al appletului are coordonata `O(0, 0)`, iar pe orizontală avem semi-axa `Ox` și pe verticală `Oy`. Dimensiunea appletului se fixează folosind atributele `width` (lățime = `Ox`) și `height` (înălțime = `Oy`) ale marcatorului `<applet>` din fișierul HTML care va apela fișierul `byte code` asociat programului Java. Astfel, execuția programului `AplicatiaDoi` va consta în afișarea textului `Acesta este un applet` începând cu punctul de coordonate `(50, 50)` al appletului de dimensiuni `(300, 200)`.

De la apariția sa și până în prezent, Java a fost îmbunătățit continuu, transformându-se într-un mediu de dezvoltare pentru o largă varietate de aplicații, cum ar fi aplicații de tip client/server, aplicații Internet sau aplicațiile standard independente.

Cea mai recentă platformă Java este J2ME (Java2MicroEdition), care oferă suportul pentru crearea aplicațiilor care rulează pe dispozitive mici fără fir (eng. *wireless devices*), cele mai cunoscute la noi fiind telefonul mobil și pagerul.

J2ME face parte din platforma Java 2, care mai cuprinde J2EE (Java 2 Enterprise Edition) și J2SE (Java 2 Standard Edition).

Tehnologia J2ME este asemănătoare cu cea J2SE, diferențele constând în faptul că dispozitivele pentru care sunt destinate aplicațiile J2ME folosesc resurse limitate. Astfel, pentru a crea astfel de aplicații este folosit un subset de clase din pachetele `java.lang`, `java.io`, `java.util`, pachete care fac parte din J2SE. În plus, pentru J2ME au fost construite două mașini virtuale: C Virtual Machine (CVM) și K Virtual Machine (KVM). Acestea ocupă puțină memorie și necesită mult mai puține resurse decât mașina virtuală pentru tehnologiile J2EE sau J2SE.

Pentru a rula KVM, un dispozitiv fără fir trebuie să aibă anumite caracteristici hardware. Printre acestea se numără condițiile minimale:

- între 160KB (de obicei, minim 512KB) de memorie internă pentru a rula platforma Java
- procesor pe 16-biți sau 32-biți cu viteză de 25MHz
- consum redus de energie oferit adesea de o baterie
- conectare la un anumit tip de rețea adesea fără fir (eng. *wireless network*), cu o lărgime de bandă limitată la 9600bps
- rezoluția de 96x54
- adâncimea de culoare de 1 bit (adică alb/negru)

Mediul J2ME este furnizat de către Sun în cadrul pachetului Java 2 Micro Edition Wireless Toolkit (J2MEWTK). Acest pachet este disponibil pe două platforme: Microsoft Windows și Sun Solaris.

O aplicație care poate rula pe un dispozitiv „fără fir” se numește middlet. Un astfel de middlet se aseamănă ca tehnologie cu tehnologia appleturilor Java.

c) O aplicație care se execută pe partea de server este o aplicație care este rulată de către un server ca urmare a unei cereri primite de acesta, iar rezultatul este trimis programului solicitant. De cele mai multe ori acestea extind funcționalitatea respectivului server. De exemplu, servleturile sunt aplicații care se execută pe partea de server.

Un servlet este o componentă Web, scrisă în Java, care poate fi încărcată dinamic și rulată de un server Web, și care poate interacționa cu diferiți clienți folosind o implementare a paradigmei cerere/răspuns bazată pe protocolul HTTP. Servleturile extind funcționalitatea unui server, de obicei a unui server HTTP. Spre deosebire de appleturi, servleturile nu afișează o interfață grafică către client, ci doar rezultatul procesării (de obicei sub forma unui fișier HTML).

Un caz particular de servleturi îl constituie JSP-urile (*Java Server Pages*). Această tehnologie este atât o extensie a servleturilor, cât și o extensie a limbajului HTML. Prin intermediul unor taguri particulare se pot insera declarații, expresii, secvențe de cod Java etc.

Unele aplicații mai complexe au nevoie de o cantitate mare de informații structurate, iar pentru acest lucru se recurge la utilizarea bazelor de date. Conexiunea la baze de date se realizează în Java prin intermediul JDBC (*Java Database Connection*).

JDBC este o specificație API (*Application Programming Interface*) dezvoltată de Sun Microsoft care definește o interfață uniformă pentru accesarea diferitelor baze de date. JDBC este parte a platformei Java și este inclusă în pachetul JDK. Cea mai importantă funcție a JDBC-ului este posibilitatea lucrului cu instrucțiuni SQL (eng. *Structured Query Language*) și procesarea rezultatelor într-o manieră independentă și consistentă a bazelor de date. JDBC furnizează acces orientat pe obiecte la bazele de date prin definirea de clase și interfețe care reprezintă obiecte, cum ar fi conexiuni la

baze de date, instrucțiuni SQL, mulțimi rezultat, obiecte binare și caractere de dimensiuni mari, drivere de baze de date, manageri de drivere.

Aplicațiile, fie ele de sine stătătoare, pe partea de client sau server, pot stabili conexiuni la baze de date, la diverse servere din rețea, pot salva starea obiectelor pentru a le restaura la o execuție ulterioară sau pentru trimiterea în fluxuri de date. De asemenea, pot prelucra date în diferite formate: text (documente txt, html, xml, rtf etc.), grafic (2D și 3D) și video.

În zilele noastre, cele mai multe aplicații sunt concepute pentru a rula în rețele de calculatoare, comunicând prin fluxuri de date (prin socketuri) sau apeluri de metode la distanță. Informațiile sunt transmise fie prin încapsulare în obiecte, fie prin mesaje XML. Utilizând formatul XML, comunicarea se poate realiza între aplicații care sunt implementate în limbaje de programare diferite.

Unele aplicații au nevoie de executarea metodelor altor obiecte aflate pe servere la distanță. Aceste aplicații transmit parametrii metodei și primesc rezultatul returnat de respectiva metodă.

Apelul metodelor la distanță (eng. RMI – *Remote Method Invocation*) furnizează în nivelurile de rețea intermediare posibilitatea obiectelor Java, care se află pe mașini virtuale diferite, să comunice folosind apeluri de metode normale. Scopul implementării apelului metodelor la distanță în Java este furnizarea unui cadru pentru comunicarea între obiecte prin metodele lor, indiferent de localizarea lor. Aceasta înseamnă că un obiect client poate accesa un obiect având rol de server alfat pe o mașină la distanță ca și cum ar rula în aceeași mașină virtuală Java.

Pentru crearea unei clase care va fi accesibilă de la distanță, trebuie să definim o interfață ale cărei metode sunt publice. Parametrii și valorile returnate pot fi de orice tip; transferul de date este rezolvat automat de fluxul de obiecte. Clasa trebuie să implementeze această interfață (și orice altă interfață și metode necesare folosirii în scop local). Există un instrument disponibil sub JDK, numit *rmic* (eng. *remote method invocation compiler*), care generează două fișiere *.class* importante pentru realizarea apelurilor la distanță.

1.5. Mașina virtuală Java

Mașina virtuală Java este noutatea principală a limbajului Java față de toate celelalte limbaje de programare. Partea dificilă a creării codului binar Java este compilarea codului sursă pentru o mașină care nu există fizic. Această mașină este numită *mașina virtuală* Java, deoarece există doar în memorie. Apoi, interpretorul Java traduce instrucțiuni cod mașină Java în mulțimi de instrucțiuni care pot fi înțelese de mașina reală (fizică).

Fără a intra în detalii de construcție a mașinii virtuale, vom prezenta cele mai importante părți ale unei astfel de mașini care necesită simularea unui calculator real: o mulțime de regiștri, o stivă, un mediu de execuție, un ansamblu pentru eliberarea memoriei, o tabelă (eng. *pool*) de literalii constanți, un spațiu de memorie a unei metode și o mulțime de instrucțiuni.

Pentru examinarea instrucțiunilor byte code, JDK include un instrument soft numit *dezasambler de fișiere cod binar Java*. Totuși, cu acest utilitar nu se poate dezasmbla un fișier `.class` pentru a crea fișierul sursă Java. Dezasamblerul `javap` oferă doar posibilitatea de a vedea cum lucrează o clasă, cum folosește o clasă resursele sistem și pentru a verifica instrucțiunile de importare a pachetelor și dependențele.

Dacă se execută `javap` fără opțiuni, atunci se va afișa numele fișierului `.class` compilat și declarațiile abreviate pentru variabilele, metodele și clasele publice. Dacă se execută `javap` fără opțiuni și fără argumente de tip clasă, atunci se va afișa lista tuturor opțiunilor.

Exemplul 1.5.1. Comanda:

```
javap -c -p -l -s -v AplicatiaUnu
va afișa:
```

```
Compiled from AplicatiaUnu.java
class AplicatiaUnu extends java.lang.Object {
    AplicatiaUnu();
    /*      ()V      */
    /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]);
    /*      ([Ljava/lang/String;)V      */
    /* Stack=2, Locals=1, Args_size=1 */
}
```

Method AplicatiaUnu()

```
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 return
```

Line numbers for method AplicatiaUnu()

line 1: 0

Method void main(java.lang.String[])

```
0 getstatic #2 <Field java.io.PrintStream out>
3 ldc #3 <String "Vom afisa acest mesaj.">
5 invokevirtual #4 <Method void println(java.lang.String)>
8 return
```

Line numbers for method void main(java.lang.String[])

line 3: 0

line 4: 8

Opțiunile de mai sus au semnificația următoare:

- c : pentru dezasmblarea codului;
- p : pentru afișarea tuturor membrilor și claselor;

- l : pentru tipărirea numerelor liniilor și a tabelor variabilelor locale;
- s : pentru tipărirea semnăturii de tip internă;
- v : pentru afișarea mărimii stivei, precum și a numărului de variabile locale și argumente pentru metode.

Astfel, se observă că, atunci când compilăm un program Java, compilatorul traduce fiecare linie de cod de nivel înalt în mai multe linii de instrucțiuni de nivel jos (asemănătoare cu un limbaj de asamblare). În exemplul de mai sus, compilatorul inserează un constructor implicit, apoi metoda `main()`. De exemplu, instrucțiunile de asamblare de mai sus au următorul sens:

- `aload_0` : încarcă referința obiect dintr-o variabilă locală cu index specificat;
- `invokespecial` : apelează un constructor;
- `return` : metoda curentă returnează `void`;
- `getstatic` : obține un câmp static din clasa curentă;
- `ldc` : memorează o constantă în tabela de constante din clasa curentă;
- `invokevirtual` : apelează o metodă în timpul execuției (și nu în timpul compilării).

Detalii despre compilarea codului sursă java în byte code se pot găsi în lucrarea [NoS96].

1.6. Concluzii

Kitul de instalare pentru mediul de dezvoltare Java se poate obține în mod gratuit de la adresa <http://java.sun.com>. Mașina virtuală Java este noutatea principală a limbajului Java față de alte limbaje de programare orientate obiect. La instalarea limbajului Java, se va crea o mașină virtuală Java care are drept scop traducerea instrucțiunilor unui byte code în instrucțiuni mașină pentru sistemul de operare curent. Astfel, fișierele intermediare byte code pot fi copiate și executate pe orice platformă (indiferent că este Windows, Unix, Solaris sau altele).

În principiu, există trei tipuri de aplicații Java: *aplicații de sine stătătoare*, *aplicații care se execută pe partea de client* și *aplicații care se execută pe partea de server*.

Aplicațiile de sine stătătoare sunt programe Java care pot fi rulate în afara contextului unui navigator. Cea mai importantă caracteristică este încapsularea în clasa principală a unei funcții `main()`, asemănătoare cu cea folosită în C.

Appleturile sunt încărcate de la server și rulează pe mașina clientului. În general, appleturile se încapsulează într-o pagină Web și afișează o interfață grafică către client. Acestea se apelează cu un browser Web sau folosind comanda `appletviewer` pentru un fișier HTML 3.2 care conține un tag de forma `<applet>` a cărui atribut `code` este egal cu numele appletului. În versiunea HTML 4.0., `<applet>` este considerat învechit, appleturile fiind adăugate în pagini cu ajutorul marcatorului `<object>`.

Un *middlet* este o aplicație care se poate rula pe dispozitive mobile fără fir, cum ar fi pagerele și telefoanele mobile. Acestea au la dispoziție resurse limitate.

Un servlet este o componentă Web care rulează pe un server web și care interacționează cu diferiți clienți prin intermediul paradigmei cerere/răspuns a protocolului HTTP. Spre deosebire de appleturi, servleturile nu afișează o interfață grafică către client, ci doar rezultatul procesării servleturilor (de obicei sub forma unui fișier HTML).

Aplicațiile amintite pot avea conexiuni la bazele de date prin intermediul JDBC sau pot executa metode aflate la distanță cu ajutorul tehnologiei RMI.

1.7. Test grilă

Întrebarea 1.7.1. Este limbajul Java independent de platformă?

- a) Da, deoarece se va crea o mașină virtuală Java care are drept scop traducerea instrucțiunilor unui byte code în instrucțiuni mașină pentru sistemul de operare curent;
- b) Nu, deoarece este imposibil ca același program Java să fie executat pe diferite sisteme de operare fără a fi modificat.

Întrebarea 1.7.2. Cum putem inițializa pointeri în Java?

- a) folosind operatorul de adresă &;
- b) folosind operatorul de dereferențiere *;
- c) nu există pointeri în Java.

Întrebarea 1.7.3. Ce tipuri de aplicații Java există?

- a) aplicații de sine stătătoare;
- b) baze de date;
- c) aplicații care se execută pe server sau client;
- d) clienturi.

Întrebarea 1.7.4. Cum se poate executa appletul intitulat `UnApplet.class`?

- a) folosind comanda `java UnApplet`
- b) utilizând comanda `appletviewer` și un fișier HTML care conține un tag de forma `<applet code="UnApplet.class" . . . >`
- c) utilizând comanda `javac`

Întrebarea 1.7.5. Cu ce comandă putem dezambla un cod binar Java?

- a) `java -p`
- b) `javac`
- c) `javap`
- d) `rmic`

1.8. Exerciții propuse spre implementare

Exercițiul 1.8.1. Instalați mediul de dezvoltare Java de la adresa <http://java.sun.com> și rulați exemplele prezentate în secțiunile precedente.

2. Fundamentele limbajului Java

2.1. Cuvinte cheie

- fișiere sursă, pachete, definiții import, definiții de clase
- case-sensitive
- atomi lexicali (identificatori, cuvinte rezervate, literal, separatori, operatori, comentarii)
- tipuri de date (primitive, referință)
- variabile (membre, locale)
- expresii, declarații (întregi, raționali în virgulă flotantă, caracter, tablouri), conversii
- structuri de control (blocuri și instrucțiuni, expresii condiționale)

2.2. Fișiere sursă

Toate fișierele sursă Java trebuie să se termine cu extensia .java. Un fișier sursă ar trebui să conțină cel mult o definiție a unei clase de bază publice. Dacă există o clasă publică, atunci numele clasei trebuie să coincidă cu numele fișierului (bineînțeles, fără extensie). Un fișier sursă poate conține un număr nelimitat de definiții de clase nepublice.

Există trei elemente de prim rang care pot apărea în fișier. Nici unul nu este obligatoriu, însă dacă apar, atunci trebuie scrise în ordinea: declarații de pachete, instrucțiuni de includere (eng. *import*) de clase, definiții de clase.

Un *pachet* este o mulțime de programe Java (de obicei, sub forma fișierelor de cod binar Java) păstrate într-un același director. Există mai multe pachete de bază, cum ar fi: java.lang, java.applet, java.io etc. Pachetul java.lang nu necesită a fi declarat în programele Java, acesta fiind importat implicit. Ca urmare, putem avea programe Java care să nu importe explicit pachete.

Pachetele sunt utile pentru separarea claselor în vederea reutilizării lor. De obicei aceste clase nu vor mai suferi modificări majore. Spre exemplu, un programator a implementat o serie de clase pentru lucru cu numere fracționare. Acestea pot fi stocate în același director și vor putea fi reutilizate în alte aplicații prin includerea pachetului creat.

Formatul declarării unui pachet este simplu, necesitând scrierea cuvântului rezervat package urmat de numele pachetului. Numele pachetului este un șir de caractere separat prin puncte. După ce s-au creat fișierele clasă, acestea trebuie memorate într-o structură de subdirectoare care reflectă numele pachetelor. Pentru a păstra independența de platformă, Java înlocuiește simbolurile „/” din UNIX și „\” din Windows cu caracterul „.”.

Instrucțiunile de import de fișiere au o formă similară, putând importa o clasă dintr-un pachet sau un pachet întreg. Sintaxa generală în aceste cazuri este:

```
import <numePachet>.<numeClasa>;
respectiv:
import <numePachet>*;
```

Exemplul 2.2.1. Pentru a exemplifica crearea și importarea pachetelor, considerăm două fișiere plasate în directorul C:\temp\pachet\exemplu\ de pe un sistem Windows care conțin câte o clasă simplă fiecare, astfel:

```
package pachet.exemplu;

public class ClasaUnu {
    public ClasaUnu() {
        System.out.println("Suntem in ClasaUnu");
    }
}
respectiv:
```

```
package pachet.exemplu;

public class ClasaADoua {
    public ClasaADoua() {
        System.out.println("Suntem in ClasaADoua");
    }
}
```

După compilarea acestor fișiere sursă, vom crea în directorul C:\temp un alt fișier sursă care va importa aceste clase definite în pachetul pachet.exemplu. Fie acesta:

```
import pachet.exemplu.*;

public class TestPachet {
    public static void main(String[] args) {
        ClasaUnu cu = new ClasaUnu();
        ClasaADoua cad = new ClasaADoua();
    }
}
```

La compilarea acestui fișier, trebuie respectate două cerințe:

- în CLASSPATH trebuie adăugat și directorul C:\temp
- în directorul C:\temp nu trebuie să se afle fișierele ClasaUnu.java, ClasaUnu.class, respectiv ClasaADoua.java, ClasaADoua.class. Prezența unuia dintre acestea ar conduce la o eroare de compilare.

CLASSPATH este o variabilă de mediu care conține lista tuturor directoarelor în care se caută fișierele .class conținând clase utilizate de aplicațiile Java.

Astfel, la execuție se vor putea accesa clasele definite în pachetul pachet.exemplu și rezultatul va fi:

```
Suntem in ClasaUnu
Suntem in ClasaADoua
```

2.3. Atomi lexicali

Etapa de compilare a programelor Java începe cu *analizarea lexicală* a codului sursă Java. Analizarea lexicală înseamnă parcurgerea secvențială a textului identificând cuvinte acceptate de sintaxa Java, numite *atomi lexicali* (eng. *tokens*). Acești atomi lexicali sunt identificatori, cuvinte rezervate, literali, separatori și operatori ai limbajului Java. Spațiile și comentariile sunt de asemenea identificate, însă acestea nu mai sunt transmise copiei programului Java în vederea continuării procesului de compilare (analiză sintactică, semantică etc.).

2.3.1. Caractere Unicode

Programele Java pot fi scrise folosind setul de caractere ASCII și eventual codări UNICODE în notația \udddd. Astfel traducerea lexicală poate folosi orice secvență Unicode cu sens special (eng. *escape*) pentru a include orice caracter Unicode folosind doar caractere ASCII (eng. *American Standard Code for Information Interchange*). Pentru implementarea diferitelor convenții ale sistemelor actuale de descriere a numerelor de linie consistente sunt definiți terminatori de linie.

Detalii despre caracterele Unicode se găsesc la adresele:

- <http://www.unicode.org/unicode/standard/standard.html>
- <http://ietf.org/rfc/rfc2278.txt>

Limbajul Java este *case-sensitive*, adică face diferențierea între caracterele mari și mici. De exemplu, cuvântul `for` este diferit de `FOR` sau `For`.

În ceea ce privește reprezentarea textului și codificarea caracterelor, Java utilizează două feluri de reprezentări text:

- setul de caractere Unicode - pentru reprezentarea internă a caracterelor și șirurilor de caractere;
- codări UTF - pentru intrări și ieșiri.

Unicode folosește 16 biți pentru reprezentarea unui caracter. Dacă cei mai semnificativi 9 biți sunt toți 0, atunci codificarea corespunde standardului ASCII (ultimii 7 biți fiind reprezentarea caracterelor). Altfel, biții reprezintă un caracter care nu este reprezentat în ASCII pe 7 biți. Tipul `char` din Java folosește codificarea Unicode și clasa `String` conține o înșiruire de caractere Java.

Setul de caractere Unicode este suficient pentru majoritatea limbilor, dar nu și pentru limbile asiatice, care conțin prea multe caractere. Soluția acestei probleme constă în utilizarea formatului UTF (eng. *Universal Character Set Transformation Format*). Codificarea UTF folosește exact numărul dorit de biți: mai puțini pentru alfabete mici, mai mulți pentru alfabete mari.

Prin *codificare* a caracterelor înțelegem o corespondență bijectivă între mulțimea caracterelor și o mulțime de numere binare. Fiecare platformă Java are o codificare a caracterelor implicită, care este utilizată pentru interacțiunea dintre mașina Java care folosește Unicode intern și sistemul de operare peste care este instalată aceasta. Codificarea implicită a caracterelor reflectă limba și cultura locală. Fiecare codificare are un nume. De exemplu, „ISO-8859_1” înseamnă `LATIN_1`, „ISO-8859_8” este ISO Latin/Hebrew și „CP1258” este vietnameză.

Exemplul 2.3.1. Următorul program Java va returna numele codificării caracterelor de pe platforma Java curentă (de exemplu, `Cp1250`). Vom folosi metoda `getEncoding()` din clasa `InputStreamReader`.

```
import java.io.*;

public class CaractereUnicode {
    public static void main(String args[]) {
```

```
        InputStreamReader isr = new InputStreamReader(System.in);
        System.out.println(isr.getEncoding());
    }
}
```

Când are loc o operație de intrare/ieșire în Java, sistemul trebuie să cunoască codificarea caracterelor. Clasele I/O utilizează de obicei codificarea implicită locală, în caz că nu se precizează una explicită. Cu toate că codificarea implicită locală este adecvată, în aplicațiile de rețea, când se comunică într-o rețea cu un alt computer, acestea trebuie să folosească aceeași codificare. În acest caz, este recomandată cererea explicită „ISO-8859_1”. Pentru mai multe informații, se poate consulta capitolul destinat internaționalizării aplicațiilor. Toate implementările platformelor Java suportă următoarele codificări ale caracterelor:

- US-ASCII (cod ASCII american);
- ISO-8859-1 (Alfabetul Latin ISO Nr. 1);
- UTF-8 (format de transformare Unicode pe opt biți);
- UTF-16BE, UTF-16LE, UTF-16 (diferite formate de transformare Unicode pe 16 biți).

2.3.2. Traduceri lexice

Un flux de caractere Unicode de pe o linie este tradus într-o secvență de atomi lexicali Java, folosind următorii trei pași de traducere:

1. traducerea secvențelor Unicode din fluxul caracterelor Unicode în caracterul Unicode corespunzător;
2. traducerea fluxului Unicode de la pasul 1 într-un flux de caractere de intrare și terminatori de linie;
3. traducerea fluxului caracterelor de intrare și a terminatorilor de linie de la pasul 2 în secvențe de elemente de intrare Java care comprimă atomii lexicali, ce devin simbolii terminali pentru gramatică sintactică Java (spațiile și comentariile sunt eliminate).

Pentru rezolvarea ambiguităților elementelor de intrare Java (identificarea corectă a atomilor lexicali), analiza lexicală se bazează pe asocierea celui mai lung cuvânt al traducerii.

Exemplul 2.3.2. Caracterele de intrare din textul `a==b` vor fi împărțite în `a`, `==`, `b`, și nu în `a`, `=`, `=`, `b`.

Implementările Java recunosc întâi secvențele escape de la intrare, traducând caracterele ASCII care încep cu `'\u'` și sunt urmate de patru cifre hexazecimale, restul caracterelor fiind copiate fără schimbări.

Exemplul 2.3.3. Secvența `\u0064\u0066` { se traduce în secvența `do {`.

Apoi, implementările Java se divid în secvențe de caractere de intrare Unicode în linii folosind terminatori de linie. Astfel se pot determina numărul liniilor folosite

într-un program Java, precum și specificarea sfârșitului comentariilor liniare (de forma `//`). Linile sunt terminate de caracterele ASCII CR (eng. *carriage return*), LF (eng. *line feed*) sau ambele (adică CR LF). Cele două caractere CR LF se numără ca fiind un singur caracter. Caracterele escape asociate cu CR și LF sunt `'\r'` și respectiv `'\n'`.

Java definește trei tipuri de comentarii:

- Ca și în limbajele C/C++, textul ASCII dintre caracterele `/*` și `*/` se numește *comentariu tradițional*.
- Ca și în limbajul C++, textul ASCII de după caracterele `//` până la sfârșitul liniei se numește *comentariu liniar*.
- Textul ASCII dintre caracterele `/**` și `*/` se numește *comentariu de documentare*.

Un *identificator* este o secvență de lungime arbitrară (nelimitată) de litere și cifre Java în care primul este o literă Java. Un identificator nu poate fi cuvânt rezervat, literal boolean sau literal null.

Exemplul 2.3.4. `oVariabila`, `numeFunctie`, `ch`, `MititeiGustosi` și `_unu` sunt identificatori, pe când `2_doi` și `10NegriMititei` nu sunt identificatori.

Un *literal* este reprezentarea codului sursă a unei valori de tip primitiv, tip `String` sau tip `null`. Un *literal întreg* poate fi exprimat în zecimal (baza 10), hexazecimal (baza 16) sau octal (baza 8). Prefixând scrierea numărului cu `0` înseamnă că numărul este scris în octal, iar cu `0x` (sau `0X`) înseamnă că numărul este scris în hexazecimal. Un *literal în virgulă flotantă* este compus din: parte întreagă, punct zecimal, parte fracționară, un exponent și un sufix de tip. Exponentul este indicat de prezența literei `e` sau `E`. Un literal în virgulă flotantă este de tip `float` dacă este sufixat cu litera `f` sau `F`, altfel tipul său este `double` și poate fi opțional sufixat de litera `d` sau `D`. Un *literal boolean* poate fi reprezentat prin `true` și `false` și este întotdeauna de tip `boolean`. Un *literal caracter* poate fi exprimat printr-un caracter sau secvență escape încadrate între apostrofuri și este întotdeauna de tip `char`. Un *literal String* constă din zero sau mai multe caractere încadrate de ghilimele, fiecare caracter putând fi reprezentat de o secvență escape. Un literal `String` este întotdeauna de tip `String` și se referă la aceeași instanță a clasei `String`. Literalul `null` este de tip `null`.

Exemplul 2.3.5. Literalul întreg `28` din baza 10 este `034` (văzut astfel în baza 8) sau `0x1c` (văzut astfel în baza 16). Numărul `10000L` este un literal `long` (pe 8 octeți). Exemple de literal în virgulă flotantă sunt `3.573` (folosind punctul zecimal), `3.67e3` (folosind notația științifică, numărul are valoarea $3.67 \cdot 10^3 = 3670$), `3.567f` (indicând un literal `float`) și `23.75d` (indicând un literal `double`). Singurii literal booleani sunt `false` și `true`. Exemple de literal `char` sunt `'g'`, `' '`, sau exprimând cifre hexazecimale Unicode, `'\ubaba'`, `'\u3456'`, sau prin secvențe escape, `'\n'` (pentru linie nouă), `'\r'` (pentru Return), `'\t'` (pentru Tab), `'\b'` (pentru Backspace), `'\f'` (pentru Formfeed), `'\''` (pentru apostrof), `'\"'` (pentru ghilimele), `'\"'` (pentru Backslash). Exemple de literal `String` sunt `"Acesta este un sir."` sau `"\u0064\u0066"` (" (care se traduce în secvența "do {}").

Recunoașterea liniei de la intrare rezultată din procesarea escape a caracterelor de intrare și a terminatorilor de linie continuă cu reducerea la o secvență de elemente de

intrare. Literele și cifrele recunoscute de Java pot fi obținute începând cu JDK 1.1 folosind metodele statice ale clasei `Character`:

```
public static boolean isJavaIdentifierStart(char ch);
public static boolean isJavaIdentifierPart(char ch);
```

Metoda `isJavaIdentifierStart(char ch)` întoarce `true`, dacă `ch` poate fi prima literă a unui identificator Unicode (literă, `$` sau `_`). Metoda `isJavaIdentifierPart(char ch)` întoarce `true`, dacă `ch` poate fi parte a unui identificator, în afara primei poziții (literă, cifră, `$` sau `_`).

Cuvintele rezervate Java sunt: `abstract`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`, `default`, `do`, `double`, `else`, `extends`, `final`, `finally`, `float`, `for`, `goto`, `if`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`, `package`, `private`, `protected`, `public`, `return`, `short`, `static`, `super`, `switch`, `synchronized`, `this`, `throw`, `throws`, `transient`, `try`, `void`, `volatile`, `while`. Pentru a da mesaje de eroare mai bune, cuvintele (din limbajul C++) `const` și `goto` sunt rezervate în Java, chiar dacă nu sunt folosite în mod curent. Literalii booleani `true` și `false`, respectiv literalul `null` nu sunt cuvinte rezervate.

Separatorii sunt `(, )`, `{, }`, `[, ]`, `;`, `.,`, iar operatorii sunt `=`, `>`, `<`, `!`, `~`, `?:`, `==`, `<=`, `>=`, `!=`, `&&`, `||`, `++`, `--`, `+`, `-`, `*`, `/`, `&`, `|`, `^`, `%`, `<<`, `>>`, `>>=`, `+=`, `-=`, `*=`, `/=`, `&=`, `|=`, `^=`, `%=`, `<<=`, `>>=`, `>>>=`.

2.4. Tipuri de date

Java este un limbaj *puternic tipizat*, adică fiecare variabilă și expresie are un tip cunoscut în momentul compilării. Tipurile de date limitează valorile pe care o variabilă le poate memora sau pe care le poate produce o expresie, limitează operațiile suportate de aceste valori și determină înțelesul acestor operații.

În Java, există trei categorii de tipuri de date: *primitive*, *referință* și *null*. Tipurile primitive sunt tipurile *numerice* și *boolean*. Tipurile numerice sunt tipurile *integrale* (`byte`, `short`, `int`, `long` și `char`) și tipurile în *virgulă flotantă* (`float` și `double`). Tipurile referință sunt tipurile *clasă*, *interfață* și *tablou*. Un obiect în Java este o instanță creată dinamic a unui tip clasă sau a unui tablou creat dinamic. Valorile unui tip referință sunt referințe la obiecte. Toate obiectele, inclusiv tablourile, suportă metode ale clasei `Object`. Literalii `String` sunt reprezentați de obiecte `String`.

Tipul `null` este special, acesta neavând nume. Este deci imposibil să declarăm o variabilă de tip `null` sau să o convertim la o expresie de tip `null`. Referința `null` este o valoare posibilă pentru o expresie de tip `null` și poate fi convertită la orice tip referință.

Numele tipurilor sunt folosite în declarații, conversii, în expresii de creare a instanței unei clase, în expresii de creare a tablourilor și în expresii folosite de operatorul `instanceof`.

Valorile primitive nu împart starea cu alte valori primitive, deci o variabilă de tip primitiv va ține o valoare primitivă de acel tip. Valoarea unei variabile de tip primitiv poate fi schimbată doar prin operații de atribuire a acelei variabile.

Referitor la tipurile primitive de date, avem următoarele informații:

Tip	Dimensiunea reprezentării (număr de bit)	Valoarea minimă a domeniului	Valoarea maximă a domeniului
boolean	1	false	true
byte	8	$-2^7 = -128$	$2^7 - 1 = 127$
short	16	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
int	32	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
long	64	$-2^{63} = -9223372036854775808$	$2^{63} - 1 = 9223372036854775807$
char	16	$\backslash u0000 = 0$	$\backslash uffff = 65535$
float	32	$[-0x7f7fffff, -0x1]$	$[0x1, 0x7f7fffff]$
double	64	$[-0x7fffffffffffffffL, -0x1L]$	$[0x1L, 0x7fffffffffffffffL]$

Orice valoare a oricărui tip integral poate fi convertită la sau către un tip numeric. În schimb, nu există conversii între tipurile integrale și tipul boolean. Depășirea domeniului implică atribuirea unei valori care nu concordă cu realitatea matematică.

Exemplul 2.4.1. Fie secvența de cod Java:

```
int a = 1400000000, b = 1600000000;
System.out.println(a + b);
```

Aceasta va afișa -1294967296 care corespunde cu interpretarea domeniului unui tip de date cu un buffer circular. Astfel, valoarea reală de 3000000000 depășește domeniul $\{-2147483648, \dots, 0, \dots, 2147483647\}$. Cum valoarea 2147483648 corespunde cu -2147483648, rezultă că 3000000000 va fi interpretată ca fiind $-2147483648 + (3000000000 - 2147483648) = -1294967296$.

Există și depășiri de domeniu care sunt detectate încă de la compilare.

Exemplul 2.4.2. La compilarea instrucțiunii:

```
short u = 32768;
```

se obține eroarea la compilare: „possible loss of precision”. O rezolvare a acestei erori este să realizăm o conversie explicită a literalului 32768 la short. Așa cum este de așteptat, se va tipări -32768 care este o valoare eronată (ca în exemplul 2.4.1).

```
short u = (short) 32768;
System.out.println("u = " + u);
```

Tipurile float și double respectă standardul ANSI/IEEE (eng. *American National Standard Institute/Institute of Electrical and Electronics Engineers*) numărul 754/1985 referitor la valorile și operațiile în virgulă mobilă. Astfel, tipul float are domeniul de

valori între $[-3.4E+38, -1.4E-45]$ U $[1.4E-45, 3.4E+38]$ și furnizează o precizie de 6-7 cifre semnificative, pe când tipul double are domeniul de valori între $[-1.79E+308, -4.94E-324]$ U $[4.94E-324, 1.79E+308]$ și furnizează o precizie de 14-15 cifre semnificative. Clasele Float și Double pun la dispoziție constante și metode membre importante pentru domeniul de valori, astfel:

- MAX_VALUE, MIN_VALUE (pentru precizarea valorilor extreme pozitive);
- POSITIVE_INFINITY, NEGATIVE_INFINITY (pentru precizarea valorilor infinite);
- NaN (eng. *Not-a-Number*), folosită pentru rezultatul operațiilor speciale, cum ar fi împărțirea la zero.

Valorile negative ale domeniului sunt -MAX_VALUE, respectiv -MIN_VALUE. La numere zecimale, împărțirea la zero se poate preveni folosind metoda (din clasele Double sau Float):

```
public static boolean isInfinite(Double x)
care returnează true, dacă argumentul x este infinit.
```

Exemplul 2.4.3. Următoarea secvență de cod testează împărțirea la zero:

```
double x = 2, y = 0;
if (Double.isInfinite((x / y)))
    System.out.println("infinit");
else
    System.out.println((x / y));
```

Obținerea numărului NaN poate fi testată folosind metoda isNaN() din clasele Float sau Double (cu sau fără argument):

```
public static boolean isNaN(double v);
```

Situațiile care pot apărea la împărțiri între două numere zecimale sunt exprimate în tabelul:

x	y	x / y	x % y
finit	± 0.0	$\pm$ infinit	NaN
finit	$\pm$ infinit	± 0.0	valoarea lui x
± 0.0	± 0.0	NaN	NaN
$\pm$ infinit	finit	$\pm$ infinit	NaN
$\pm$ infinit	$\pm$ infinit	NaN	NaN

Exemplul 2.4.4. Folosind metoda isNaN(), putem preveni operațiile fără sens:

```
double x = 0, y = 0;
if (Double.isNaN(x / y))
    System.out.println("operatie care conduce la obtinerea "+
        "unui ne-numar !");
```

Exemplul 2.4.5. Putem testa dacă valoarea unui număr zecimal este prea mică astfel:

```
if (Math.abs(a) < 0.1E-20f) . . .
```

Exemplul 2.4.6. Putem testa dacă aplicarea funcției radical de ordin 2 este corectă (argumentul este pozitiv):

```
float a = 0, b = 1, d = -5;
double radical = Math.sqrt(d);
if (!Double.isNaN(radical)) {
    double radacina = (-b + radical) / (2.0f * a);
    System.out.println(radacina);
} else {
    System.out.println("d este negativ sau este NaN");
}
```

O operație în virgulă flotantă care depășește domeniul produce infinit cu semn sau zero infinit. De asemenea, operațiile aritmetice care nu au rezultat definit matematic returnează NaN. Operațiile numerice care au un operand NaN returnează NaN. Deoarece NaN nu induce o ordine, operațiile de comparație care conțin NaN returnează false, cu excepția lui != care returnează true. Rezultatul operației în virgulă mobilă este rotunjit la numărul cel mai apropiat conform cu precizia tipului de date respectiv.

Exemplul 2.4.7. Următorul program Java pune în evidență depășirea domeniului valorilor în virgulă flotantă, precum și rotunjirea făcută în unele situații:

```
public class NumereInVirgulaMobila {
    public static void main(String args[]) {
        // depasire a domeniului catre plus infinit
        double infinitMare = 1e308;
        System.out.println(infinitMare * 10);
        // depasire a domeniului catre zero pozitiv
        double infinitMic = 1e-323;
        System.out.println(infinitMic / 10);
        // rotunjire a unui rezultat inexact de tip float
        // Matematic, ar fi trebuit sa se afiseze 1.0
        float rezultatInexactFloat = 1.0f / 41;
        System.out.println(rezultatInexactFloat * 41);
        // rotunjire a unui rezultat inexact de tip double
        // Matematic, ar fi trebuit sa se afiseze 1.0
        double rezultatInexactDouble = 1.0f / 49;
        System.out.println(rezultatInexactDouble * 49);
    }
}
```

Rezultatul afișat de execuția acestui program este:

```
Infinity
```

```
0.0
0.99999994
0.9999999795109034
```

Tipul boolean este reprezentat prin două valori posibile indicate de literalii true și false.

Continuăm cu prezentarea tipului referință. Un obiect este o instanță a unei clase sau un tablou. Valorile referință sunt pointeri la aceste obiecte sau sunt egale cu referința specială null (care nu se referă la nici un obiect). O instanță a unei clase este explicit creată folosind o expresie de creare a unei instanțe sau prin apelul metodei newInstance() din clasa Class. Un tablou este explicit creat printr-o expresie de creare a unui tablou. Un obiect de tip tablou este creat implicit când se evaluează o expresie de inițializare a unui tablou. Aceasta poate apărea când este inițializată o clasă sau interfață, când este creată o nouă instanță a unei clase sau când se execută o instrucțiune de declarare a unei variabile locale. Declararea și definirea claselor și tablourilor vor fi prezentate în detaliu în capitolele următoare.

Exemplul 2.4.8. Următorul exemplu pune în evidență crearea unor instanțe pentru o clasă:

```
class Exemplu {
    int x;
    // constructor implicit
    Exemplu() { System.out.println("constructor implicit"); }
    // constructor explicit
    Exemplu(int x) { this.x = x; }
    // instanta creata explicit la momentul initializarii clasei
    static Exemplu primulX = new Exemplu(1);
    // metoda de actualizare a campului x pentru obiectul curent
    public void setX(int x) {
        this.x = x;
    }
    // metoda necesara pentru afisarea explicita a unei
    // instanțe din clasa Exemplu
    public String toString() {
        return "x = " + x;
    }
} // sfarsitul definitiei clasei Exemplu

class DiferiteInstante {
    public static void main(String[] args) {
        System.out.println("primulX: " + Exemplu.primulX);
        // apelul constructorului implicit
        Exemplu obiectDoi = null;
        // creare explicita folosind newInstance()
        try {
```

```

    obiectDoi = (Exemplu)Class.forName("Exemplu").
        newInstance();
}
catch (Exception e) {
    System.out.println(e);
}
System.out.println("obiectDoi: " + obiectDoi);
obiectDoi.setX(2);
System.out.println("obiectDoi: " + obiectDoi);
// crearea unui obiect folosind operatorul new
Exemplu obiectTrei = new Exemplu(3);
System.out.println("obiectTrei: " + obiectTrei);
} // sfarsitul definitiei clasei DiferiteInstante

```

Execuția acestui program Java conduce la ieșirea:

```

primulX: x = 1
constructor implicit
obiectDoi: x = 0
obiectDoi: x = 2
obiectTrei: x = 3

```

Clasa `Object` este o superclasă pentru toate clasele Java. O variabilă de tip `Object` poate ține o referință a oricărui obiect, fie că este o instanță a unei clase sau un tablou. Toate tipurile clasă și tablou moștenesc metodele clasei `Object`.

Instanțele clasei `String` reprezintă secvențe de caractere Unicode. Un obiect `String` are o valoare constantă (care nu se schimbă în timpul execuției programului). Literalii `String` sunt referințe la instanțe ale clasei `String`. Operatorul de concatenare a șirurilor + creează implicit un nou obiect `String`.

Două tipuri referință sunt de același tip dacă sunt tipuri tablou ale căror componente sunt de același tip sau sunt tipuri clasă sau interfață, sunt încărcate de același încărcător de clasă și au același nume (se mai spune că sunt în aceeași clasă sau interfață).

Exemplul 2.4.9. Exemplul de mai jos pune în evidență utilizarea tipurilor de date Java:

```

import java.util.Random;
import java.io.*;

public class Zaruri {

    public static void main(String[] arg) {
        // instantieri
        Random r = new Random();
        int[] zar = new int[2];
        for (int k = 0; k < 10; k++) {

```

```

        for(int i = 0; i < 2; i++) {
            zar[i] = (int)(r.nextDouble() * 6) + 1;
        }
        System.out.println("\nZarurile sunt: " + zar[0] + " " +
            zar[1]);
        if (zar[0] == zar[1])
            System.out.println("Ati castigat! Zarurile sunt" +
                "egale!");
        else
            System.out.println("Ati pierdut! Zarurile nu sunt" +
                "egale!");
        // asteptam apasarea tastei ENTER
        System.out.print("Apasati tasta ENTER pentru a" +
            "continua...");
        try {
            // citim un caracter de la tastatura
            System.in.read();
            // eliminam celelate caractere citite
            System.in.skip(System.in.available());
        } catch (IOException e) {
            System.out.println(e);
        }
        System.out.println("Aplicatia s-a terminat!");
    } // sfarsitul definitiei clasei Zaruri

```

Tipurile de date sunt folosite în următoarele declarații:

1. Tipuri importate: de exemplu, tipul `Random` importat de `java.util.Random` al pachetului `java.util`;
2. Tipuri ale câmpurilor clasei `Zaruri`: de exemplu, tipul `Button` importat din `java.awt.Button` este utilizat pentru un câmp al clasei `Zaruri`; la fel, tipul primitiv `boolean`;
3. Tipuri folosite pentru parametrii metodelor: de exemplu, tipul `String[]` este utilizat de metoda `main()`;
4. Tipuri folosite pentru variabile locale: tipul primitiv `int` este folosit pentru variabila locală `i` a metodei `main()`;
5. Tipuri folosite pentru crearea unei instanțe a unei clase: tipul `Random` pentru crearea variabilei `r`;
6. Tipuri folosite pentru crearea tablourilor: tipul `int` este folosit pentru crearea tabloului `zar` cu două elemente;
7. Tipuri folosite pentru conversii explicite: conversia la tipul `int` este utilizată în cea de-a doua instrucțiune `for`, pentru a ne asigura că valoarea zarurilor sunt numere întregi între 1 și 6.

2.5. Expresii și operatori

Un program Java se bazează în principal pe *evaluarea expresiilor*, fie prin *efectele laterale* (asignarea de valori către variabile în timpul evaluării expresiilor), fie pentru *valorile returnate* de acestea (utilizate ca argumente sau operanzi în alte expresii) sau pentru *afectarea secvenței de execuție* în instrucțiuni.

Când se evaluează o expresie într-un program Java, rezultatul întors poate fi o variabilă, o valoare sau nimic (eng. *void*). Deoarece expresiile pot conține asignări interne, operatori de incrementare/decrementare și apeluri de metode, rezultă că evaluarea unei expresii poate produce efecte laterale. O expresie poate să întoarcă *void* în cazul apelurilor de metode care întorc *void*, caz în care expresia este instrucțiune și nu servește la transmiterea valorii sale. Fiecare expresie apare într-o declarație a unui tip clasă sau interfață care poate fi inițializator de câmp, inițializator static, declarație a unui constructor sau în codul unei metode.

Dacă o expresie este o variabilă sau o valoare, atunci expresia trebuie să aibă tipul cunoscut încă de la compilare. Ca și la variabile, valoarea unei expresii este compatibilă cu tipul expresiei. Astfel, valoarea unei expresii al cărei tip este *T* poate fi asignată unei variabile de tip *T*.

Dacă tipul unei expresii este un tip primitiv, atunci valoarea expresiei este de același tip primitiv. Dacă tipul expresiei este un tip referință, atunci clasa și valoarea obiectului referențiat nu se cunosc la momentul compilării, ci al execuției.

Pentru a da definiția formală a *expresiei*, mai întâi definim noțiunea de *expresie primară*. Expresiile primare sunt literalii, accesul la câmpuri și tablouri, apeluri de metode, expresii parantetizate. Un *operator* cu *n* argumente este o funcție cu *n* argumente care se mai numesc și *operanzi*. Utilizând o manieră recursivă de descriere, o *expresie compusă* (sau, simplu, *expresie*) poate fi expresie primară, o expresie care folosește operatori unari, binari și ternari sau o expresie de conversie. De exemplu, operatorul *+* este unar și binar, operatorul condițional *?:* este ternar etc.

În continuare, vom prezenta *instrucțiunea de asignare*. O instrucțiune de asignare are sintaxa:

```
<numeVariabila> = <expresie> ;
```

unde *=* se numește *operator de asignare*. Semantica acestei instrucțiuni este simplă: se evaluează *<expresie>* și se memorează valoarea acesteia la adresa variabilei *<numeVariabila>*.

Exemplul 2.5.1. În urma execuției instrucțiunilor de asignare:

```
int a = 4, b = 7;
int c = a + b;
```

se obține valoarea 11 pentru variabila *c*. Acestea se pot scrie într-o „singură instrucțiune” astfel:

```
int a, b, c = (a = 4) + (b = 7);
```

Expresiile (*a = 4*) și (*b = 7*) întorc valoarea părții drepte, adică 4 și, respectiv, 7. Astfel, se face asignarea tuturor celor trei variabile într-o manieră compactată.

O altă versiune a instrucțiunii de asignare este:

```
<numeVariabila> <operator>= <expresie> ;
```

unde *<operator>* poate fi: *+*, *-*, ***, */*, *%*, *<<*, *>>*, *>>>*, *&*, *|* și *^*, între *<operator>* și *=* nefiind spații (în caz contrar, se obține eroare la compilare). Aceasta este o „prescurtare” a instrucțiunii:

```
<numeVariabila> = <numeVariabila> <operator> (<expresie>);
```

Secvența execuției operatorilor cu aceeași precedență dintr-o instrucțiune este determinată de *proprietatea de asociativitate*. Astfel, în Java există două tipuri de asociativități: *stângă* și *dreaptă*. Spunem că *<op>* este operator stâng asociativ dacă pentru orice *x*, *y*, *z* avem *x <op> y <op> z = (x <op> y) <op> z*. Spunem că *<op>* este operator drept asociativ dacă pentru orice *x*, *y*, *z* avem *x <op> y <op> z = x <op> (y <op> z)*.

Se mai spune că operatorii stâng asociativi se evaluează „de la stânga la dreapta”, iar cei drept asociativi „de la dreapta la stânga”. De exemplu, operatorul *=*, precum și cei de forma *<operator>=* sunt drept asociativi.

Exemplul 2.5.2. Fie instrucțiunile de asignare:

```
int a = 1, b, c = 3, d;
a += b = c += d = 5;
```

Cea de-a doua linie se poate rescrie astfel:

```
a += (b = (c += (d = 5)));
```

Astfel, se obțin prin efecte laterale valorile 9, 8, 8, 5 pentru variabilele *a*, *b*, *c* și *d*.

În Java, operanzii se evaluează într-o ordine predefinită, și anume de la stânga la dreapta. De obicei, fiecare expresie are cel mult un efect lateral și codul nu este dependent de excepția care apare drept consecință a evaluării expresiilor de la stânga la dreapta. În cazul operatorilor binari, operandul din stânga este complet evaluat înaintea evaluării oricărei părți a operandului din dreapta.

Exemplul 2.5.3. Execuția următorului program Java:

```
class EvaluareStanga {
    public static void main(String[] args) {
        int a = 5;
        int b = (a = 4) * a;
        System.out.println(b);
    }
}
```

va afișa 16, deoarece se evaluează mai întâi subexpresia (*a = 4*) care întoarce 4. Mai mult, valoarea lui *a* este 4, și nu 5 cum era inițial. Astfel, valoarea lui *b* este *4 * 4*, adică 16.

Continuăm cu prezentarea *operatorilor*, urmând ca în secțiunea următoare să prezentăm noțiunea de variabilă. Vom prezenta operatorii după tipurile operanzilor pe care-i pot avea.

Începem cu operatorii care lucrează cu operanzi de tipuri integrale. În afara constructorilor și metodelor din clasele `Integer`, `Long` și `Character`, avem următorii operatori:

1. *Operatorii de comparație*, care întorc o valoare de tip `boolean`: `<`, `<=`, `>`, `>=`, `==`, `!=`
2. *Operatorii numerici*, care întorc o valoare de tip `int` sau `long`: `+`, `-` (unari și binari), `*`, `/`, `%` (multiplicativi), `++`, `--` (incrementare și decrementare, formele prefixată și postfixată), `<<`, `>>`, `>>>` (deplasare cu și fără semn), `~` (complement față de 2), `&`, `|`, `^` (și, sau inclusiv, sau exclusiv pe biți)
3. *Operatorul condițional* `?` :
4. *Operatorul de conversie explicită*, care poate converti o valoare integrală la o valoare de un tip numeric specificat
5. *Operatorul de concatenare a șirurilor*, care, în cazul unui operand de tip `String` și unul de tip integral, va face conversia celui integral la `String`, creând astfel un nou obiect `String` egal cu concatenarea acestora

Dacă un operator întreg (diferit de cel de deplasare) are un operand de tip `long`, atunci operația este efectuată folosind o precizie de 64 de biți și convertind celălalt operand la tipul `long`, valoarea returnată de operatorul numeric fiind de tip `long`. În caz contrar, operația este efectuată folosind o precizie de 32 de biți, convertind operanzii la tipul `int`, rezultatul operatorului numeric fiind de tip `int`. Operatorii întregi predefiniți nu indică depășirea domeniului (Exemplul 2.4.1), singurele excepții de tip `ArithmeticException` care pot apărea se pot datora împărțirii la zero în cazul operatorilor `/` și `%`.

Exemplul 2.5.4. Următorul program va scoate în evidență apariția unei excepții aritmetice:

```
public class ExceptiiNumerice {
    public static void main(String args[]) {
        int a = 3, b = 0;
        try {
            System.out.println("a / b = " + a / b);
        }
        catch (ArithmeticException e) {
            System.out.println("Nu este permisa impartirea la zero:"
                + e.getMessage());
        }
    }
} // sfarsitul definitiei clasei ExceptiiNumerice
```

Operatorii de comparație sunt stâng asociativi, însă în marea majoritate a cazurilor necesită paranteze.

Exemplul 2.5.5. Expresia `1 < 2 < 3` se interpretează `(1 < 2) < 3`, care se evaluează la `true < 3` și conduce la o eroare de compilare, deoarece operatorul `<` nu lucrează cu operanzi de tip `boolean`.

În cazul operatorilor de comparație, tipul fiecărui operand trebuie să fie tip numeric primitiv, altfel se obține o eroare de compilare. Evaluarea expresiilor care conțin operatori de comparație se realizează în mod natural. De exemplu, pentru operatorul `<` avem `<operandUnu> < <operandDoi>` se evaluează la `true` dacă valoarea lui `<operandUnu>` este mai mică decât valoarea lui `<operandDoi>`.

Pentru ceilalți operatori, lucrurile decurg asemănător (`<=` este „mai mic sau egal”, `>` este „mai mare”, `>=` este „mai mare sau egal”, `==` este „egal cu” și `!=` este „diferit de”). Operatorii `==` și `!=` se mai numesc și *operatori de egalitate* și se deosebesc de ceilalți operatori prin faptul că au prioritatea mai mică.

Exemplu 2.5.6. Datorită priorității mici a operatorului `==`, expresia `1 < 2 == 3 < 4` se parsează ca fiind `(1 < 2) == (3 < 4)`. Aceasta se evaluează la `true == true`, care returnează `true`.

Tipul unei expresii de egalitate este întotdeauna `boolean`. În toate cazurile, `a != b` este echivalent semantic cu `!(a == b)`. Operatorii de egalitate sunt comutativi dacă expresiile operanzilor nu au efecte laterale.

Referitor la operatorul unar de adunare `+`, putem spune că tipul expresiei operandului `+` trebuie să fie un tip numeric primitiv, altfel obținem o eroare la compilare. Tipul expresiei plus unar va fi tipul convertit la compilare, iar rezultatul expresiei plus unar nu este o variabilă, ci o valoare (chiar dacă rezultatul expresiei operand este o variabilă).

În ceea ce privește operatorul unar de scădere, acesta este asemănător cu cel unar de adunare. Deoarece Java utilizează reprezentarea complement față de doi pentru întregi și domeniul valorilor complement față de doi nu este simetric, rezultă că negarea valorii maxime negative pentru tipurile `int` și `long` iese din domeniu, afișând valoarea maximă negativă (fără a arunca vreo excepție). Pentru toate valorile întregi, `-x` este egal cu `(~x) + 1`.

Exemplul 2.5.7. Execuția următorului subprogram Java:

```
System.out.println(Integer.MAX_VALUE + " " + -Integer.MAX_VALUE);
System.out.println(Long.MAX_VALUE + " " + -Long.MAX_VALUE);
```

va afișa:

```
2147483647 -2147483647
9223372036854775807 -9223372036854775807
```

Operatorii `+`, `-` se mai numesc *operatori de adunare* și se aplică operanzilor numerici de același tip, producând suma, respectiv diferența acestora. Adunarea este o operație comutativă (pentru orice `a`, `b` avem `a + b = b + a`) dacă expresiile operanzilor nu au efecte laterale. Adunarea întregilor este asociativă când toți operanzii sunt de același tip (nu este cazul adunării numerelor în virgulă flotantă). Dacă o

adunare întregă depășește domeniul de valori, atunci rezultatul va corespunde unei sume matematice reprezentate în formatul complement față de doi și, eventual, va avea alt semn (Exemplul 2.4.1). Operatorul binar de scădere – folosește în definiția sa operatorii binari de adunare și unar de scădere astfel: pentru orice a, b avem $a - b = b + (-a)$.

Operatorii $*$, $/$ și $\%$ se numesc *operatori multiplicativi* și se aplică operanzilor numerici de același tip, executând produsul (înmulțirea), raportul (împărțirea), respectiv restul împărțirii acestora. Acești operatori au aceeași precedență și sunt stâng asociativi. Tipul fiecărui operand al unui operator multiplicativ trebuie să aparțină unui tip numeric primitiv, altfel obținem o eroare de compilare. Înmulțirea este o operație comutativă dacă expresiile operanzilor nu au efecte laterale. Înmulțirea întregilor este asociativă când toți operanzii sunt de același tip (nu este cazul înmulțirii numerelor în virgulă flotantă). Dacă o înmulțire întregă depășește domeniul de valori, atunci rezultatul va corespunde unui produs matematic reprezentat în formatul complement față de doi și, eventual, va avea alt semn (Exemplul 2.4.1).

Pentru operanzi întregi, operatorul $/$ întoarce partea întreagă a împărțirii primului operand la cel de-al doilea. Astfel, notând $a / b = c$ (c fiind câtul dintre a și b) și $a \% b = d$ (d fiind restul împărțirii lui a la b), putem spune că c are semnul produsului dintre a și b , iar d are semnul lui a . Acest lucru previne utilizarea greșită a teoremei împărțirii cu rest din matematică ($a = b * c + d$, unde $0 \leq d < b$). Dacă valoarea lui b este 0, atunci se aruncă o excepție `ArithmeticException`.

Exemplul 2.5.8. Avem: $7 / 2 = 3$, $7 \% 2 = 1$, $(-7) / 2 = -3$, $(-7) \% 2 = -1$, $7 / (-2) = -3$, $7 \% (-2) = 1$, $(-7) / (-2) = 3$, $(-7) \% (-2) = -1$.

Referitor la operatorii (unari, de incrementare/decrementare $++$, $--$), aceștia pot apărea în formă postfixată (operatorul este la sfârșit) sau prefixată (operatorul este la început). Rezultatul expresiei unare trebuie să fie o variabilă a unui tip numeric, altfel apare o eroare de compilare. Tipul expresiei de incrementare/decrementare este identic cu tipul variabilei. În cazul operatorilor postfixați, de exemplu $a++$ sau $a--$, se va aduna/scădea valoarea 1 la vechea valoare a variabilei a , aceasta modificându-și valoarea cu 1. Valoarea expresiei $a++$ sau $a--$ este vechea valoare a variabilei a (adică înaintea modificării ei). Ca și la forma postfixată, în cazul operatorilor prefixați, de exemplu $++a$ sau $--a$, se va aduna/scădea valoarea 1 la vechea valoare a variabilei a , aceasta modificându-și valoarea cu 1. Valoarea expresiei $++a$ sau $--a$ este noua valoare a variabilei a (adică modificarea ei).

Rezultatul expresiei de incrementare/decrementare este o valoare și nu o variabilă. În plus, este evident că operanzii operatorilor $++/--$ nu pot fi constanți (declarați final sau chiar literali). Astfel, nu putem scrie `int i = 4++`.

Exemplul 2.5.9. Iată câteva exemple de utilizare a operatorilor unari de incrementare/decrementare:

```
int i = 5;
System.out.println(++i + " " + ++i + " " + ++i);
```

Datorită evaluării de la stânga la dreapta a expresiei argument din metoda `println()`, rezultă că se va afișa:

```
6 7 8
```

Continuăm cu liniile de cod Java:

```
int i = 3, j = 7, k = ++i + j--;
System.out.println(k);
```

În urma execuției acestor ultime două linii se va afișa 11.

Continuăm cu prezentarea operatorilor de deplasare. Astfel, $<<$ este operatorul de deplasare stânga, $>>$ este operatorul de deplasare dreapta cu semn, iar $>>>$ este operatorul de deplasare dreapta fără semn. Aceștia sunt stâng asociativi, operandul din dreapta specificând numărul de poziții cu care se deplasează. Tipul fiecărui operand al unui operator de deplasare trebuie să fie de tip integral, altfel apare o eroare de compilare. Operandul din dreapta este redus modulo 32, dacă operandul din stânga este (sau poate fi convertit la) `int`, sau modulo 64, dacă operandul din stânga este (sau poate fi convertit la) `long`. Valoarea expresiei $a << b$, unde b este întreg pozitiv, este egală cu valoarea lui a deplasată cu b poziții spre stânga, care este echivalentă cu a înmulțit cu 2^b . Valoarea expresiei $a >> b$ este egală cu valoarea lui a deplasată cu b poziții spre dreapta, care pentru numere pozitive este echivalentă cu partea întreagă a lui a împărțit cu 2^b . Valoarea lui $a >>> b$ este egală cu valoarea lui a deplasată la dreapta cu b poziții. Dacă a este pozitiv, atunci rezultatul coincide cu $a >> b$. Dacă a este negativ, rezultatul lui $a >>> b$ este egal cu expresia $(a >> b) + (2 << \sim s)$ dacă tipul operandului stâng este `int` sau $(a >> b) + (2L << \sim s)$ dacă tipul operandului stâng este `long`. Termenul care se adună (este vorba de $(2 << \sim s)$, respectiv $(2L << \sim s)$) are rolul de a inhiba propagarea din stânga a simbolurilor 1.

Exemplul 2.5.10. Dacă avem:

```
int a = -5;
int b = a >>> 1;
```

atunci b nu este un număr negativ, cum se întâmplă în cazul operatorului de deplasare dreapta $>>$ (prezent de altfel și în limbajul C), și va avea valoarea 2147483645.

Exemplul 2.5.11. Ne așteptăm ca expresia `-64 >>> 4` să aibă valoarea 12, deoarece `-64` are reprezentarea binară 11000000, iar după deplasarea la dreapta cu 4 biți să obținem 00001100. Din păcate, lucrurile nu stau deloc așa. Mai întâi se face conversia lui `-64` (care este literal de tip `byte`) la `int`. Deci acesta va fi în memorie (în binar):

```
11111111 11111111 11111111 11000000
```

Ne deplasăm la dreapta cu 4 biți și obținem:

```
00001111 11111111 11111111 11111100
```

Se observă că ultimul octet diferă la cele două reprezentări (11000000 față de 11111100). Astfel, valoarea întoarsă de expresia `-64 >>> 4` va fi de tip `int` și va avea valoarea 268435452. De altfel, în Java nu putem scrie instrucțiuni de genul:

```
byte b = -64 >>> 4;
```

deoarece obținem eroare la compilare (eng. *Explicit cast to int is required*). Explicația se datorează faptului că valoarea `-64` se memorează pe mai mult de un octet și deci la conversia implicită s-ar pierde din precizie.

Continuăm cu prezentarea operatorului unar de complementariere. Tipul expresiei operand a operatorului `~` trebuie să fie un tip integral, altfel apare o eroare de compilare. Tipurile `short`, `byte` se convertesc implicit la `int`, astfel că expresia `~a` are tipul `int` sau `long`. Regulele de complementariere aplicate alfabetului $\{0, 1\}$ sunt: $\sim 0 = 1$ și $\sim 1 = 0$. Acestea se pot generaliza la cuvinte de lungime 32, respectiv 64, astfel: $\sim a_1 a_2 \dots a_n = \sim a_1 \sim a_2 \dots \sim a_n$.

Exemplul 2.5.12. Este clar că aplicând operatorul de complementaritate unui număr pozitiv, acesta devine negativ și reciproc. Acest lucru se datorează faptului că și bitul de semn devine contrar. Astfel:

```
~ 11100111 10111000 10101001 11000010 = 00011000 01000111
01010110 00111101
```

Valoarea numerelor pozitive din memoria calculatorului se deduce din teorema fundamentală a aritmeticii ($a_1 a_2 \dots a_n(d) = a_1 \cdot d^{n-1} + a_2 \cdot d^{n-2} + \dots + a_n$). Pentru numere negative, se folosește în plus regula $\sim x = (-x) - 1$, de unde deducem că:

$$\sim x = \sim x + 1.$$

Exemplul 2.5.13. Numerele pozitive au primul bit (se mai numește bit de semn) egal cu 0. Astfel, $a = 00011000\ 01000111\ 01010110\ 00111101 = 1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^9 + 2^{10} + 2^{12} + 2^{14} + 2^{16} + 2^{17} + 2^{18} + 2^{22} + 2^{27} + 2^{28} = 407328317$. Numerele negative au primul bit egal cu 1, iar valoarea lor o vom calcula cu formula $x = -(\sim x + 1)$. Astfel, pentru calculul lui $b = 11011001\ 01001101\ 01011110\ 10110101$, vom calcula valoarea complementarului său, care este un număr pozitiv: $00100110\ 10110010\ 10100001\ 01001010 = 2^1 + 2^3 + 2^6 + 2^8 + 2^{13} + 2^{15} + 2^{17} + 2^{20} + 2^{21} + 2^{23} + 2^{25} + 2^{26} + 2^{29} = 649240906$. Adunăm acum o unitate și apoi schimbăm semnul, obținând astfel $b = -649240907$. Pentru obținerea formei binare a unui literal integral, putem utiliza o metodă din clasa `Integer` (sau `Long`), și anume (începând cu JDK versiunea 1.0.2):

```
public static String toBinaryString(int i);
```

care întoarce reprezentarea din memorie a numărului `i` (în baza doi, fără a pune în evidență zerourile nesemnificative). Astfel, executând programul Java de mai jos:

```
public class FormaBinara {
    public static void main(String args[]) {
        System.out.println(Integer.toBinaryString(407328317));
    }
}
```

```
System.out.println(Integer.toBinaryString(-649240907));
}
```

facem verificarea calculelor matematice de mai sus:

```
11000010001110101011000111101
11011001010011010101111010110101
```

Continuăm acum cu operatorii pe biți „și” `&`, „sau inclusiv” `|`, „sau exclusiv” `^`. Acești operatori sunt stâng asociativi, dar au precedente diferite, operatorul `&` având cea mai mare prioritate și operatorul `|` având cea mai mică prioritate. În plus, acești operatori sunt comutativi dacă expresiile operandilor nu au efecte laterale. Când ambii operanzi sunt de tip integral, atunci aceștia se convertesc la `int` sau `long`, după caz. Aceștia au definițiile matematice obișnuite, astfel:

1. $a \& b = 1$ dacă și numai dacă $a = 1$ și $b = 1$;
2. $a | b = 1$ dacă și numai dacă $a = 1$ sau (inclusiv) $b = 1$;
3. $a ^ b = 1$ dacă și numai dacă $a = 1$ sau $b = 1$ și $a \neq b$.

Definiția 2 se poate da echivalent: $a | b = 0$ dacă și numai dacă $a = 0$ și $b = 0$. La fel, definiția 3 se poate rescrie astfel: $a ^ b = 1$ dacă și numai dacă $a + b = 1$.

Exemplul 2.5.14. Fie instrucțiunile:

```
int a = 3333, b = 7777;
```

Iată care sunt reprezentările în memorie, respectiv valorile corespunzătoare în baza 10:

Expresie	Reprezentare	Valoare (baza zece)
a	00000000 00000000 00001101 00000101	3333
b	00000000 00000000 00011110 01100001	7777
a&b	00000000 00000000 00001100 00000001	3073
a^b	00000000 00000000 00010011 01100100	4964
a b	00000000 00000000 00011111 01100101	8037
~(a b)	00000000 00000000 11100000 10011010	-8038
~a&~b	00000000 00000000 11100000 10011010	-8038

Se observă că se verifică relațiile lui De Morgan, cum ar fi $\sim(a|b) = \sim a \& \sim b$. Putem considera și operații pentru numere scrise în baza 16, cum ar fi $0 \times \text{ff}00 \& 0 \times \text{f}0\text{f}0 = 0 \times \text{f}000$, $0 \times \text{ff}00 | 0 \times \text{f}0\text{f}0 = 0 \times \text{ff}\text{ff}$ și $0 \times \text{ff}00 ^ 0 \times \text{f}0\text{f}0 = 0 \times \text{ff}\text{f}0$.

Continuăm cu operatorul condițional `?`: care apare în expresii (condiționale) de forma: `<expresieBooleana> ? <expresie> : <expresie>`, unde `<expresie>` poate fi la rândul ei expresie condițională. Operatorul condițional este drept asociativ, spre deosebire de majoritatea operatorilor. `<expresieBooleana>` trebuie să fie de tip boolean, altfel apare o eroare de compilare. Operatorul condițional poate fi

folosit pentru a alege dintre operanzii al doilea și al treilea de tip numeric, de tip boolean, de tip referință sau tip null. Toate celelalte cazuri conduc la o eroare de compilare. Valoarea întoarsă de o expresie condițională este determinată la execuție astfel: mai întâi se evaluează expresia booleană; dacă aceasta se evaluează la true, atunci se evaluează operandul al doilea a cărui valoare se returnează; în caz contrar (dacă aceasta se evaluează la false), se returnează valoarea celui de-al treilea operand. Deoarece operatorul condițional nu evaluează toți operanzii, se mai spune că acesta este un operator „leneș” (eng. *lazy*).

Exemplul 2.5.15. Expresia `1<2?3:4<5?6:7` se interpretează astfel: `(1<2)?3:(4<5)?6:7`, care se evaluează la 3.

Referitor la tipul întors de expresia condițională, avem următoarele reguli:

1. dacă al doilea și al treilea operand au același tip (poate fi și tipul null), atunci acesta va fi tipul returnat de expresia condițională;
2. altfel, dacă al doilea și al treilea operand au tip numeric, atunci avem subcazurile:
 - a. dacă unul din operanzi este de tip byte și celălalt este de tip short, atunci tipul expresiei condiționale este short;
 - b. dacă unul dintre operanzi este de tip T, unde T poate fi byte, short sau char, și celălalt operand este o expresie constantă de tip int a cărei valoare este reprezentabilă în tipul T, atunci tipul expresiei condiționale este T;
 - c. altfel, se fac conversii implicite la compilare tipurilor celor de-al doilea și al treilea operand și acesta va fi tipul returnat de expresia condițională.
3. dacă cel de-al doilea sau al treilea operand este de tip null și tipul celui alt operand este referință, atunci tipul expresiei condiționale este acel tip referință;
4. dacă cel de-al doilea și al treilea operand sunt tipuri referință diferite, atunci se încearcă o conversie de atribuire (eng. *assignment conversion*) a unui tip la celălalt tip. În caz de succes, acesta va fi tipul expresiei condiționale, altfel se va obține o eroare de compilare.

Exemplul 2.5.16. Fie declarația `byte b=25;`. Tipul întors de expresia condițională `(1<2)?b:40` este byte, deoarece se aplică regula 2b de mai sus. Expresia `(1<2)?"abc":40` va returna o eroare la compilare, deoarece nu este posibilă o conversie de atribuire între String și int.

Continuăm cu prezentarea succintă a conversiilor numerice. Tipul unei expresii cast este tipul al cărui nume apare între paranteze (se mai numește *operator cast*). Rezultatul unei expresii cast nu este o variabilă, ci o valoare, chiar dacă rezultatul expresiei operand este o variabilă. De exemplu, o expresie cast convertește la execuție o valoare de tip numeric la o valoare similară a altui tip numeric. Prezentarea în detaliu a operatorului de conversie va fi făcută în secțiunea 2.8 (Conversii).

Exemplul 2.5.17. Fie subprogramul Java:

```
short sh1 = 5;
int in1 = sh1;
System.out.println(in1);
```

```
int in2 = 20000;
short sh2 = (short)in2;
System.out.println(sh2);
```

Se observă că de la tipuri „mai mici” (care se reprezintă pe mai puțini octeți) la tipuri „mai mari” nu este necesară prezența conversiei explicite (cum este cazul conversiei de la short la int). În schimb, dacă se încearcă atribuirea unei variabile de tip mai mare unei variabile de tip mai mic, atunci se obține eroare la compilare care semnalează pierderea posibilă a unor date (eng. *possible loss of data*). Astfel, este necesară prezența conversiei explicite care în situația exemplului nostru este (short). Deși nu mai obținem eroare la compilare, la execuție se pot obține rezultate eronate (datorită trunchierii datelor prin conversia explicită). De exemplu, execuția subprogramului:

```
int in2 = 200000;
short sh2 = (short)in2;
System.out.println(sh2);
```

va afișa valoarea 3392 obținută prin „înlăturarea” primilor doi octeți care reprezintă în memorie valoarea 200000 (3392 reprezintă, de fapt, restul împărțirii lui 200000 la 32768).

Continuăm cu ultimul operator referitor la tipuri întregale, și anume operatorul de concatenare a șirurilor. Orice tip Java poate fi convertit la tipul String. O valoare x a tipului primitiv T este mai întâi convertită la o valoare referință apelând la clasele corespunzătoare tipului T (eng. *wrapper*) având ca parametru pe x:

1. pentru T = char, se folosește `new Character(x)`
2. pentru T = byte, short, int, se folosește `new Integer(x)`
3. pentru T = long, se folosește `new Long(x)`

Pentru obiecte primitive, implementarea poate optimiza crearea unui obiect „wrapper” prin convertirea directă a unui tip primitiv la un obiect de tip String.

Exemplul 2.5.18. De exemplu, dorim să citim dintr-un șir de caractere (numit sir) un întreg (numit numar):

```
Integer obiectIntreg = new Integer(sir);
int numar = obiectIntreg.intValue();
```

Am utilizat unul din constructorii clasei Integer și am creat un obiect din această clasă. Metoda `intValue()` va întoarce valoarea întreagă conținută de obiectIntreg. Deoarece clasa Integer conține metoda:

```
public static int parseInt(String s);
```

se poate evita crearea unui obiect din clasa Integer, astfel:

```
int numar = Integer.parseInt(sir);
```

Implementarea poate alege între conversie și concatenare într-un pas pentru evitarea creării unui obiect `String` intermediar. De altfel, implementarea Java folosește clasa `StringBuffer` pentru a reduce numărul de obiecte `String` create pentru evaluarea unei expresii. Operatorul `+` este stâng asociativ, deci `a+b+c` este privit ca `(a+b)+c`.

Exemplul 2.5.19. Fie subprogramul Java, care pune în evidență asociativitatea operatorului `+`:

```
String s1 = "a" + 2 + 3, s2 = 2 + 3 + "b";
System.out.println("s1 = " + s1 + ", s2 = " + s2);
```

În cazul evaluării lui `s1`, ambii operatori `+` se referă la obiecte `String` și astfel se convertesc 2, respectiv 3, la "2", respectiv "3". În concluzie, `s1` va fi evaluat la "a23". În ceea ce-l privește pe `s2`, primul operator `+` se referă la adunarea a două numere întregi, iar al doilea operator `+` se referă la concatenarea șirurilor având al doilea argument din clasa `String`. Astfel, `s2` se va evalua la "5b" (și nu la "23b"!).

Continuăm cu operatorii care lucrează cu operanzi de tip virgulă flotantă. În afara constructorilor și metodelor din clasele `Float`, `Double` și `Math`, avem următorii operatori:

1. *Operatorii de comparație*, care întorc o valoare de tip `boolean`: `<`, `<=`, `>`, `>=`, `==`, `!=`
2. *Operatorii numerici*, care întorc o valoare de tip `float` sau `double`: `+`, `-` (unari și binari), `*`, `/`, `%` (multiplicativi), `++`, `--` (incrementare și decrementare, formele prefixată și postfixată)
3. *Operatorul condițional ? :*
4. *Operatorul de conversie explicită*, care poate converti o valoare în virgulă mobilă la o valoare de un tip numeric specificat
5. *Operatorul de concatenare a șirurilor*, care în cazul unui operand de tip `String` și unul de tip în virgulă flotantă, va face conversia operandului în virgulă mobilă la `String`, creând astfel un nou obiect `String` egal cu concatenarea acestora

Dacă un operator numeric în virgulă mobilă are un operand de tip `double`, atunci operația este efectuată folosind aritmetica în virgulă mobilă și 64 de biți și convertind celălalt operand la tipul `double`, valoarea returnată de operatorul numeric fiind tot de tip `double`. În caz contrar, operația este efectuată folosind o precizie de 32 de biți, convertind operanzii la tipul `float`, și astfel rezultatul operatorului numeric este de tip `float`. Operatorii în virgulă flotantă nu indică depășirea domeniului și nici nu indică excepții.

Referitor la operatorii în virgulă flotantă, începem cu operatorii de comparație. În plus față de operatorii integrali, putem spune că rezultatul unei comparații este următorul:

1. Dacă unul din operanzi este `NaN`, atunci rezultatul este `false`;
2. Toate valorile diferite de `NaN` sunt ordonate, cu minus infinit mai mic decât toate valorile finite și plus infinit mai mare decât toate valorile finite;
3. Zero negativ și pozitiv sunt considerate egale.

Exemplul 2.5.20. Expresia `-0.0<0.0` se evaluează la `false`, iar `-0.0<=0.0` se evaluează la `true`. Clasa `Math` conține două metode statice care întorc minimul, respectiv maximum a două numere în virgulă flotantă:

```
public static double min(double a, double b);
public static double max(double a, double b);
Astfel, execuția subprogramului Java:
System.out.println(Math.min(-0.0,0.0));
System.out.println(Math.max(-0.0,0.0));
```

va afișa:

```
-0.0
0.0
```

Referitor la operatorii în virgulă flotantă `==` și `!=`, distingem în plus față de aceeași operatori integrali următoarele (conform standardului american IEEE 754):

1. Dacă un operand este `NaN`, atunci rezultatul lui `==` este `false`, iar rezultatul lui `!=` este `true`;
2. Zero pozitiv și zero negativ sunt egale;
3. Altfel, două valori distincte în virgulă mobilă se consideră inegale, deci rezultatul lui `==` este `false`.

Putem spune că valoarea întoarsă de expresia `x!=x` este `true` dacă și numai dacă `x` este `NaN`. Metoda `isNaN()` din clasele `Float` și `Double` se poate utiliza pentru prevenirea operațiilor fără sens (care conduc la obținerea lui `NaN`).

Exemplul 2.5.21. Valoarea întoarsă de expresia `-0.0==0.0` este `true`. La fel și expresia `Double.POSITIVE_INFINITY != Double.NEGATIVE_INFINITY` întoarce `true`.

Referitor la operatorii `+`, `-` unari, putem spune în plus că la operanzii în virgulă flotantă negația nu mai este la fel ca scăderea din zero. Iată câteva cazuri speciale pentru operatorul unar `-`:

1. dacă operandul este `NaN`, atunci rezultatul este `NaN`;
2. dacă operandul este plus sau minus infinit, atunci rezultatul este minus sau respectiv plus infinit;
3. dacă operandul este zero negativ sau pozitiv, atunci rezultatul va fi zero pozitiv sau respectiv negativ.

Exemplul 2.5.22. Execuția următorului subprogram Java:

```
double x = +0.0;
System.out.println(0.0-x + " " + -x);
double y = -0.0;
System.out.println(0.0-y + " " + -y);
```

va afișa:

```
0.0 -0.0
0.0 0.0
```

Continuăm cu operatorii +, - binari. Adunarea în virgulă flotantă nu este asociativă (Exemplul 2.5.23). Rezultatul unei adunări în virgulă flotantă se determină după următoarele reguli:

1. Dacă unul dintre operanzi este NaN, atunci rezultatul este NaN;
2. Suma a două infinități de semn opus este NaN (operație fără sens);
3. Suma a două infinități de același semn este respectivul infinit;
4. Suma dintre un infinit și o valoare finită este respectivul infinit;
5. Suma a două zerouri de semn opus este zero pozitiv;
6. Suma a două zerouri de același semn este zero de acel semn;
7. Suma dintre zero și o valoare finită este acea valoare finită;
8. Suma dintre două numere finite opuse ca semn, dar egale în valoare absolută, este zero pozitiv;
9. În restul cazurilor, se calculează suma. Dacă suma a două numere este prea mare, atunci se întoarce infinit (spre deosebire de numerele întregi). Dacă suma a două numere este prea mică pentru reprezentare, se întoarce zero apropiat ca semn. Altfel, suma este rotunjită către cea mai apropiată valoare reprezentabilă.

Exemplul 2.5.23. Neasociativitatea numerelor în virgulă flotantă se poate vedea în următorul subprogram Java, unde suma a două numere în virgulă flotantă depășește domeniul:

```
float a = 2E38f, b = - a;
System.out.println((a + a) + b);
System.out.println(a + (a + b));
```

care va afișa:

```
Infinity
2.0E38
```

Ca și la numere întregi, scăderea a două numere în virgulă flotantă este ca o adunare cu opusul numărului. Cu toate acestea, scăderea din zero nu este identică cu negația (Exemplul 2.5.22). Operațiile de adunare/scădere nu vor arunca niciodată excepții de execuție.

Continuăm cu operatorul de multiplicare (înmulțire) * pentru numere în virgulă flotantă. Multiplicarea în virgulă flotantă nu este asociativă. Rezultatul unei înmulțiri în virgulă flotantă este dat de regulile:

1. Dacă unul dintre operanzi este NaN, atunci rezultatul este NaN;
2. Dacă rezultatul nu este NaN, atunci semnul rezultatului este pozitiv dacă ambii operanzi au același semn sau negativ dacă operanzii au semne diferite (*regula semnelor*);
3. Înmulțirea dintre infinit și zero este NaN (operație fără sens);
4. Înmulțirea dintre infinit și o valoare finită este infinit cu semnul determinat cu regula semnelor;
5. În celelalte cazuri, se calculează produsul celor doi operanzi. Dacă produsul este prea mare, atunci se întoarce infinit. Dacă produsul este prea mic, atunci se întoarce zero. Altfel, produsul este rotunjit către cea mai apropiată valoare reprezentabilă.

Operația de înmulțire nu va arunca niciodată excepții în timpul execuției programelor Java.

Continuăm cu operatorul de împărțire / pentru numere în virgulă flotantă. Împărțirea în virgulă flotantă nu este asociativă. Rezultatul unei împărțiri în virgulă flotantă este dat de regulile:

1. Dacă unul dintre operanzi este NaN, atunci rezultatul este NaN;
2. Dacă rezultatul nu este NaN, atunci semnul rezultatului este pozitiv dacă ambii operanzi au același semn sau negativ dacă operanzii au semne diferite (*regula semnelor*);
3. Împărțirea dintre infinit și infinit este NaN (operație fără sens);
4. Împărțirea dintre infinit și o valoare finită este infinit cu semnul determinat cu regula semnelor;
5. Împărțirea dintre o valoare finită și infinit este zero cu semnul determinat cu regula semnelor;
6. Împărțirea dintre o valoare finită și zero este semnalizată prin aruncarea la execuție a unei excepții;
7. În celelalte cazuri, se calculează raportul celor doi operanzi. Dacă raportul este prea mare, atunci se întoarce infinit. Dacă raportul este prea mic, atunci se întoarce zero. Altfel, raportul este rotunjit către cea mai apropiată valoare reprezentabilă.

În cazul împărțirii cu zero se va arunca excepția `ArithmeticException` în timpul execuției programelor Java. Exemplele 2.4.3, 2.4.4, 2.4.5 și 2.4.6 se referă la operațiile de înmulțire/împărțire pentru numere în virgulă flotantă.

Continuăm cu prezentarea operatorului % pentru operanzi în virgulă flotantă. Prin definiție, $a \% b = a - b * [a/b]$, unde $[x]$ reprezintă partea întreagă inferioară a lui x . În schimb, regulile IEEE 754 diferă prin considerarea celui mai apropiat întreg în locul părții întregi. Astfel, `Math.IEEEremainder(a,b) = a - b * Math rint(a/b)`. Rezultatul unei operații % în virgulă flotantă este determinat de regulile:

1. Dacă unul dintre operanzi este NaN, atunci rezultatul este NaN;
2. Dacă rezultatul nu este NaN, atunci semnul rezultatului este egal cu semnul operandului care se împarte;
3. Dacă operandul care se împarte este infinit sau operandul la care se împarte este zero, atunci rezultatul este NaN;
4. Dacă operandul care se împarte este finit și operandul la care se împarte este infinit, atunci rezultatul este operandul care se împarte;
5. Dacă operandul care se împarte este zero și operandul la care se împarte este finit, atunci rezultatul este operandul care se împarte;
6. În celelalte cazuri, se folosește formula de mai sus ($a \% b = a - b * [a/b]$).

Evaluarea operatorului % pentru operanzi în virgulă mobilă nu aruncă niciodată o excepție la execuție, chiar dacă operandul din dreapta este zero.

Exemplul 2.5.24. Următorul subprogram Java pune în evidență asemănarea și deosebirea dintre comportarea lui % și a lui Math.IEEEremainder():

```
System.out.println(9.8 % 4.8);
System.out.println(3.2 % 4.8);
System.out.println(9.8 - 4.8 * Math.floor(9.8 / 4.8));
System.out.println(3.2 - 4.8 * Math.floor(3.2 / 4.8));
System.out.println(Math.IEEEremainder(9.8, 4.8));
System.out.println(Math.IEEEremainder(3.2, 4.8));
System.out.println(9.8 - 4.8 * Math rint(9.8 / 4.8));
System.out.println(3.2 - 4.8 * Math rint(3.2 / 4.8));
```

va afișa la execuție:

```
0.200000000000000107
3.2
0.200000000000000107
3.2
0.200000000000000107
-1.5999999999999996
0.200000000000000107
-1.5999999999999996
```

Referitor la operatorii ++ și -- pentru operanzii în virgulă mobilă, aceștia sunt similari cu cei integrali.

Exemplul 2.5.25. Următorul subprogram Java:

```
double i = 3.52, j = 7.84, k = ++i + j--;
System.out.println(k);
```

va afișa 12.36, iar ca efect lateral i devine 4.52 și j devine 6.84.

Referitor la operatorul condițional care poate avea al doilea și al treilea operand de tip float sau double, lucrurile sunt similare cu operanzii integrali.

Exemplul 2.5.26. Fie subprogramul Java:

```
double d = 2.35;
int i = 7;
System.out.println(4<3?d:i);
```

Se va afișa 7.0, și nu 7. Tipul rezultatului expresiei condiționale trebuie să fie complet determinat în momentul compilării și, folosind regulile conversiei implicite la compilare, acest tip este double. Așadar, valoarea este tipărită în formatul în virgulă flotantă.

Operatorul de conversie explicită pentru numere în virgulă mobilă este similar cu cel de la numere integrale.

Exemplul 2.5.27. Execuția subprogramului Java de mai jos:

```
int i1 = 5;
float f1 = i1;
System.out.println(f1);
float f2 = 23.56f;
int i2 = (int)f2;
System.out.println(i2);
```

va afișa:

```
5.0
23
```

Continuăm cu ultimul operator referitor la tipuri în virgulă flotantă, și anume operatorul de concatenare șiruri. Similar cu operatorul + pentru tipuri integrale, tipurile float și double pot fi convertite la tipul String. O valoare x a tipului primitiv T este mai întâi convertită la o valoare referință apelând la clasele corespunzătoare tipului T (eng. *wrapper*) având ca parametru pe x:

1. pentru T = float se utilizează new Float(x)
2. pentru T = double se utilizează new Double(x)

Pentru obiecte primitive, implementarea poate optimiza crearea unui obiect „wrapper” prin convertirea directă a unui tip primitiv la un obiect de tip String. Citirea unui număr în virgulă flotantă este puțin mai complicată decât cele integrale (Exemplul 2.5.18), deoarece nu există analogie directă a lui parseInt() pentru numere în virgulă flotantă. De exemplu, pentru numere de tip float, se poate proceda astfel:

1. convertim șirul de caractere citit la un obiect de tip Float folosind metoda valueOf();
2. convertim valoarea în virgulă mobilă reprezentată de obiectul Float la o variabilă de tip float folosind metoda floatValue().

Exemplul 2.5.28. Următorul cod Java inițializează variabila val la o valoare în virgulă mobilă citită:

```
Float temporar = Float.valueOf(campFloat.getText());
float val = temporar.floatValue();
```

Se putea crea obiectul din clasa Float folosind unul dintre constructorii clasei Float, astfel:

```
Float temporar = new Float(campFloat.getText());
float val = temporar.floatValue();
```

Pentru variabile de tip double, se procedează similar ca la float.

Operatorul de concatenare + dintre un șir și un număr în virgulă flotantă este similar cu operatorul de concatenare + dintre un șir și un număr integral.

În continuare, prezentăm operatorii care lucrează cu operanzi de tip boolean:

1. *Operatorii relaționali* == și !=
2. *Operatorul de complement logic* !
3. *Operatorii logici* (și, sau exclusiv, sau inclusiv): &, ^, |
4. *Operatorii condiționali* (și, sau): &&, ||
5. *Operatorul condițional* ? :
6. *Operatorul de concatenare șiruri*, care în cazul unui operand de tip String și al unui de tip boolean, va face conversia operandului boolean la String, creând astfel un nou obiect String egal cu concatenarea acestora

Expresiile booleene determină controlul fluxului în instrucțiunile if, while, do, for, precum și pe cel al operatorului condițional. Primul operand al operatorului condițional poate fi doar expresie booleană. Un întreg x poate fi convertit la o valoare booleană b astfel: dacă x este diferit de zero, atunci b este true, altfel b este false.

Operatorii de egalitate booleeni sunt asociativi. Rezultatul evaluării expresiei a==b este true dacă operandii a și b sunt egali (cu true sau false). În caz contrar, rezultatul întors este false. Rezultatul evaluării expresiei a!=b este true dacă operandii a și b sunt diferiți (unul este true și celălalt false). În caz contrar, rezultatul întors este false.

Tipul expresiei operand a operatorului unar ! trebuie să fie boolean, altfel apare o eroare de compilare (și tipul întors de expresia operatorului de complement logic unar este boolean). La execuție, valoarea expresiei !a este true dacă operandul a se evaluează la false și false dacă operandul a se evaluează la true.

Referitor la operatorii logici în care operandii sunt de tip boolean, lucrurile sunt foarte simple. Tipul întors de expresiile a&b, a^b și a|b este boolean. Semantic, avem:

- a&b întoarce valoarea true, dacă valorile lui a și b sunt true; altfel, rezultatul este false;
- a^b întoarce valoarea true, dacă valorile lui a și b sunt distincte; altfel, rezultatul este false;
- a|b întoarce valoarea false, dacă valorile lui a și b sunt false; altfel, rezultatul este true.

Referitor la operatorii && și ||, acestea sunt variante performante (și „leneșe” în același timp) ale operatorilor & și |. Astfel, operatorul && este echivalent cu &, cu deosebirea că evaluează operandul drept numai dacă valoarea operandului din stânga este true. Este stâng asociativ și fiecare operand al lui && trebuie să fie de tip boolean, altfel apare o eroare de compilare. Tipul returnat de expresia condițională && este boolean. Dacă valoarea operandului a din expresia a&&b este false, atunci operandul b nu se mai evaluează, valoarea întoarsă de expresia a&&b fiind false.

Similar, operatorul || este echivalent cu |, cu deosebirea că evaluează operandul drept numai dacă valoarea operandului din stânga este false. Este stâng asociativ și fiecare operand al lui || trebuie să fie de tip boolean, altfel apare o eroare de compilare. Tipul returnat de expresia condițională || este boolean. Dacă valoarea operandului a din expresia a||b este true, atunci operandul b nu se mai evaluează, valoarea întoarsă de expresia a||b fiind true.

Exemplul 2.5.29. Execuția următorului subprogram Java:

```
int x = 2;
System.out.println((4 < 3) && (x++ < 4));
System.out.println(x);
```

va afișa:

```
false
2
```

deoarece valoarea operandului stâng (4<3) este false și atunci nu se mai evaluează operandul din dreapta operatorului leneș && care ar fi condus ca efect lateral la incrementarea lui x. Schimbând ordinea celor doi operanzi, se obține valoarea 3 pentru x:

```
System.out.println((x++ < 4) && (4 < 3));
```

Operatorul condițional ? : în care operandii sunt booleeni este similar cu cel în care operandii sunt de tip numeric. În expresia a?b:c, dacă operandul b este de tip boolean, atunci și c este de tip boolean, și reciproc.

Exemplul 2.5.30. Expresia condițională (1<2)?5:(3<4) va conduce la o eroare de compilare, deoarece nu poate converti o valoare numerică la una booleană.

Continuăm cu ultimul operator referitor la tipul boolean, și anume operatorul de concatenare șiruri. Orice tip Java poate fi convertit la tipul String. O valoare x a tipului primitiv boolean este mai întâi convertită la o valoare referință apelând la clasele corespunzătoare tipului boolean având ca parametru pe x astfel: se folosește new Boolean(x).

Exemplul 2.5.31. Folosind un constructor al clasei Boolean și rulând programul:

```
Boolean b = new Boolean(1<2);
System.out.println("adevarat ca " + b);
```

se va afișa mesajul „adevarat ca true”. Acest cod se poate rescrie transparent pentru programator astfel:

```
System.out.println("adevarat ca " + (1<2));
```

Referitor la tipul referință, expresiile primare sunt:

1. this
2. Crearea unei instanțe a unei clase (folosind cuvântul rezervat new);
3. Accesul la câmpurile unui obiect, prin nume calificat sau expresie de acces de câmp;
4. Apelul metodelor.

Operatorii care lucrează cu referințele obiectelor sunt:

1. Operatorul de conversie explicită
2. Operatorul de concatenare a șirurilor +, care în cazul unui operand de tip String și al unui de referință, va face conversia operandului referință prin apelul unei

metode `toString()`, creând astfel un nou obiect `String` egal cu concatenarea acestora

3. Operatorul `instanceof`
4. Operatorii de egalitate de referință `==` și `!=`
5. Operatorul condițional `?` :

Atât expresiile primare, cât și operatorii care lucrează cu tipul referință vor fi prezentați în Capitolul 3. Tot în categoria expresiilor primare intră și accesul la elementele unui tablou (tablourile vor fi prezentate tot în Capitolul 3).

2.6. Variabile

O *variabilă* este o locație de memorie care are un tip asociat (se mai numește tip verificat la compilare). O variabilă conține o valoare care este compatibilă cu tipul variabilei. Valoarea unei variabile se poate schimba în urma unei asignări sau a operatorilor `++` și `--`. Compatibilitatea valorilor implicite și toate asignările unei variabile sunt verificate în timpul compilării, cu excepția tabloului, când verificarea se face în timpul execuției programului Java.

Formal, o *variabilă* este un 5-uplu de forma:

Nume	Tip	Listă Modificatori	Valoare	Adresă
------	-----	--------------------	---------	--------

unde *Nume* este un identificator, *Tip* reprezintă tipul variabilei și este definit în Secțiunea 2.4, *Listă Modificatori* reprezintă lista modificatorilor unei variabile, *Valoare* reprezintă valoarea variabilei și *Adresă* reprezintă adresa acesteia. Modificatorii de acces Java sunt: `public`, `protected`, `implicit` (eng. *friendly*), `private`, iar ceilalți (care nu sunt de acces) sunt `final`, `abstract`, `static`, `native`, `transient`, `synchronized` și `volatile`. Vom reveni asupra acestor modificatori în capitolele ulterioare. Fiecare variabilă are o adresă, variabilele primitive păstrând valoarea lor în stiva de valori (eng. *Stack Memory*), iar variabilele referință (obiectele și tablourile) își păstrează o referință în stiva de valori, iar valoarea obiectului în ansamblul de valori (eng. *Heap Memory*).

Astfel, declararea unei variabile are sintaxa generală (secțiunea 2.7.):

```
[<ListaModificatori>] <Tip> <NumeVariabila>;
```

unde `<ListaModificatori>` poate conține modificatorii de mai sus, în orice ordine (cu anumite restricții), `<Tip>` reprezintă tipul variabilei și `<NumeVariabila>` este un identificator pentru numele variabilei. Restricțiile din `<ListaModificatori>` se referă la contextul apariției unui modifier. De exemplu, o variabilă nu poate fi declarată `private` și `public` în același timp. Variabilele primitive nu au asociată o listă de modificatori, aceasta fiind prezentă doar pentru variabilele obiect.

O variabilă de tip primitiv poate ține o valoare care este de acel tip primitiv. O valoare de tip referință poate ține o referință `null` sau o referință la orice obiect a cărui clasă este compatibilă cu tipul variabilei. Cu alte cuvinte, o variabilă a clasei C

poate ține o referință `null` sau o referință la o instanță a clasei C sau a unei subclase a lui C. O variabilă de tip interfață poate ține o referință `null` sau o referință la orice instanță a oricărei clase care implementează acea interfață. Dacă T este un tip primitiv, atunci o variabilă de tip *tablou de T-uri* poate ține o referință `null` sau o referință la orice tablou de tip *tablou de T-uri*. Dacă T este un tip referință, atunci o variabilă de tip *tablou de T-uri* poate ține o referință `null` sau o referință la orice tablou de tip *tablou de S-uri*, unde tipul S se poate converti la tipul T. O variabilă de tip `Object` poate ține o referință `null` sau o referință la orice obiect, indiferent dacă este instanță a unei clase sau tablou.

Exemplu 2.6.1. Pentru a obține o reprezentare a adresei interne a unei variabile obiect vom apela metoda `hashCode()` din clasa `Object`:

```
public int hashCode();
```

care returnează valoarea codului hash a unui obiect. De fiecare dată când este apelată pentru același obiect în timpul execuției unui program, metoda `hashCode()` returnează același întreg. Mai mult, dacă două obiecte sunt egale conform cu metoda `equals()`, atunci apelul metodei `hashCode()` aplicat acestor două obiecte furnizează același întreg. Acest întreg nu rămâne însă același de la o execuție la alta a aceluiași program.

Pentru a obține numele clasei care a instanțiat obiectul, vom folosi metoda `getClass()` din clasa `Class`:

```
public final Class getClass();
```

care returnează clasa obiectului în momentul execuției. Această metodă se poate folosi împreună cu metoda `getName()` din clasa `Class`:

```
public String getName();
```

care returnează numele complet al tipului (clasei, interfeței, șirului sau primitivei), reprezentată de obiect de tip `Class`, ca un `String`.

```
public class AdreseVariabile {
    public static void main(String args[]) {
        AdreseVariabile x = new AdreseVariabile();
        System.out.println("Tipul lui x = " + x.getClass().getName() +
            ", valoarea lui x = " + x + ", adresa lui x = " +
            x.hashCode());
        byte[] y = new byte[10];
        System.out.println("Tipul lui y = " + y.getClass().getName() +
            ", valoarea lui y[0] = " + y[0] + ", adresa y = " +
            y.hashCode());
    }
} // sfarsitul definitiei clasei AdreseVariabile
```

La execuția acestui program se obțin tipul, valoarea și adresa din memorie ale variabilelor `x` și `y`:

```
Tipul lui x = AdreseVariabile, valoarea lui
```

`x = AdreseVariabile@720eeb`, adresa lui `x = 7474923`
 Tipul lui `y = [B, valoarea lui y[0] = 0, adresa y = 3242435`

De remarcat că valoarea lui `x` este compusă din numele clasei care l-a instanțiat urmat de adresa variabilei în baza 16 separată prin caracterul '@'. Cu alte cuvinte, `720eeb(16) = 7474923(10)`. Pentru variabile de tip tablou, numele clasei este codificat prin caracterul '[' (care sugerează ideea de index) și, în general, prima literă a clasei wrapper asociate (B pentru byte, excepții fiind Z pentru boolean, J pentru long, L urmat de numele clasei sau interfeței pentru variabile clasă sau respectiv interfață).

Exemplul 2.6.2. Pentru declarația următoare:

```
public static final int k = 4;
```

putem spune că variabila (atribut al unei clase) `k` este de tip `int` și că este inițializată cu valoarea 4. În plus, modificatorii de acces `public`, `static` și `final` precizează că variabila `k` este publică (accesibilă de oriunde), este statică (variabilă de clasă) și constantă (nu i se poate modifica valoarea). Evident, ordinea de apariție a modificatorilor `public`, `static` și `final` se poate schimba.

În Java, există șapte tipuri de variabile:

1. *Variabila clasă* este un atribut (câmp) într-o declarație de clasă folosind cuvântul rezervat `static` sau într-o declarație de interfață (cu sau fără cuvântul rezervat `static`). O variabilă clasă este creată când este încărcată clasa (deci nu neapărat la declararea unei instanțe a clasei) și este inițializată cu valoarea implicită. Variabila clasă își încetează existența când este descărcată clasa sau interfața, după ce finalizarea clasei sau interfeței este completă.
2. *Variabila instanță* este un atribut dintr-o declarație de clasă fără folosirea cuvântului rezervat `static`. Dacă clasa `C` are un atribut `A` care este o variabilă instanță, atunci se va crea și inițializa cu valoare implicită o nouă instanță a variabilei `A` corespunzătoare fiecărui obiect nou creat al clasei `C` sau oricărei subclase a clasei `C`. Variabila instanță își încetează existența când obiectul corespunzător atributului nu mai este referențiat, după ce finalizarea clasei este completă.
3. *Componentele unui tablou* sunt variabile (fără nume) create și inițializate cu valori implicite de fiecare dată când un nou obiect tablou este creat. Componentele tabloului își încetează existența când tabloul nu mai este referențiat.
4. *Variabila parametru a unei metode* este valoarea argumentului trimis către o metodă. Pentru fiecare parametru din declarația metodei, în momentul apelului metodei se creează o nouă variabilă parametru care este inițializată cu valoarea argumentului corespunzător apelului metodei (în cazul variabilelor de tip primitiv se consideră valoarea acestora, iar în cazul variabilelor tablou și referință se consideră adresa acestora). Variabila parametru își încetează existența când se termină execuția metodei respective.
5. *Variabila parametru a unui constructor* este valoarea argumentului trimis către un constructor. Pentru fiecare parametru din declarația constructorului, se creează o variabilă parametru de fiecare dată când se apelează implicit sau

explicit constructorul. Noua variabilă este inițializată cu valoarea argumentului corespunzător al apelului constructorului. Variabila parametru a constructorului își încetează existența când se termină execuția constructorului respectiv.

6. *Variabila parametru de tratare a excepțiilor* este creată de fiecare dată când este prinsă o excepție cu o clauză `catch` a unei instrucțiuni `try`. Noua variabilă este inițializată cu obiectul actual asociat cu acea excepție. Variabila parametru de tratare a excepțiilor își încetează existența când se termină execuția blocului asociat clauzei `catch` respective.
7. *Variabila locală* este declarată de o instrucțiune de declarare a variabilei locale. De fiecare dată când fluxul de control intră într-un bloc sau instrucțiune `for`, se creează o nouă variabilă pentru fiecare variabilă locală declarată. Spre deosebire de celelalte categorii de variabile, variabilele locale nu sunt inițializate implicit, deci se va depista eroare la compilare dacă se încearcă accesarea unei variabile locale care nu a fost inițializată explicit. Variabila locală își încetează existența când se termină execuția blocului sau instrucțiunii `for` asociate.

Exemplul 2.6.3. Următorul program Java pune în evidență șase categorii de variabile care pot apărea, un exemplu de folosire a variabilelor de tip tablou este 2.4.9. Menționăm că variabilele de tip tablou vor fi prezentate în Capitolul 3:

```
import java.io.*;

public class CitiriIntregi {
    public static void main(String args[]) {
        int numar = 0;
        BufferedReader in;
        String linie;
        try {
            in = new BufferedReader(new InputStreamReader(System.in),
                                   1);
            while (true) {
                System.out.println("Dati numarul " +
                                   Radical.numarDeNumere);
                linie = in.readLine();
                try {
                    numar = Integer.parseInt(linie);
                }
                catch (NumberFormatException f) {
                    System.out.println("Nu ati tastat numar intreg");
                }
                Radical obiect = new Radical(numar);
                System.out.println("Radical din " + numar +
                                   " este " + obiect.radacinaPatrata());
                System.out.println("Continuam ? (tastati s pentru
                                   stop)");
            }
        }
    }
}
```

```

        linie = in.readLine();
        if (linie.equals("s") || linie.equals("S"))
            break;
    }
    in.close();
}
catch (IOException e) {
    System.out.println("Citire gresita de la tastatura " + e);
}
} // de la metoda main()
} // sfarsitul definitiei clasei CitiriIntregi

class Radical {
    static int numarDeNumere = 1;
    int nr;

    Radical (int nr) {
        this.nr = nr;
        numarDeNumere++;
    }

    double radacinaPatrata() {
        return Math.sqrt(nr);
    }
} // sfarsitul definitiei clasei Radical

```

Variabilele utilizate în programul de mai sus sunt:

1. **Variabilă clasă:** `numarDeNumere` este o variabilă statică și are drept scop numărarea instanțelor clasei `Radical`;
2. **Variabilă instanță:** `nr` este o variabilă atribut din declarația clasei `Radical`;
3. **Componentele unui tablou:** nu avem variabile de tip tablou;
4. **Variabilă parametru a unei metode:** `linie` este argumentul trimis către metoda `parseInt()` din clasa `Integer`;
5. **Variabilă parametru a unui constructor:** `numar` este argumentul trimis către constructorul clasei `Rational`;
6. **Variabile parametri de tratare a excepțiilor:** `f` și `e` sunt două variabile care tratează excepțiile `NumberFormatException`, respectiv `IOException`;
7. **Variabile locale:** `numar`, `in`, `linie` sunt locale metodei `main()`.

Din punct de vedere al tipului variabilelor, putem spune că `numar`, `numarDeNumere` și `nr` aparțin tipului primitiv `int`, celelalte variabile fiind de tip referință, și anume: `linie` din clasa `String`, `in` din clasa `BufferedReader`, obiect din clasa `Radical`.

2.7. Declarații și inițializări

Începem cu prezentarea instrucțiunilor de declarare a unei variabile locale. Sintaxa generală este:

```

<Tip> <numeVariabilă1> [= <Expresie>] [, <numeVariabilă2>
[= <Expresie>]] ... [, <numeVariabilăN> [= <Expresie>]];

```

unde `<Tip>` se numește tipul variabilelor al căror nume este `<numeVariabilă1>`, `<numeVariabilă2>`, ..., `<numeVariabilăN>` (acestea pot fi și nume de tablouri, caz în care se adaugă paranteze `[,]`). Parantezele pătrate `([])` semnifică variabile de tip tablou (vom vedea în Exemplul 2.7.4. că acestea pot apărea atât înaintea, cât și după numele variabilei). `<Expresie>` reprezintă o expresie de inițializare care se poate converti la tipul `<Tip>`, altfel se obține eroare la compilare.

Fiecare instrucțiune de declarare a unei variabile locale este conținută într-un bloc. Instrucțiunile de declarare a variabilelor locale pot fi intercalate printre alte instrucțiuni ale blocului. O declarare de variabilă locală poate apărea și într-o instrucțiune `for`, caz în care se execută la fel ca o instrucțiune de declarare a unei variabile locale.

O instrucțiune de declarare a unei variabile locale este o instrucțiune executabilă. La execuție, declaratorii sunt procesați de la stânga la dreapta. Dacă un declarator are o expresie de inițializare, atunci expresia este evaluată și valoarea se asignează variabilei. Dacă declaratorul nu are o expresie de inițializare, atunci compilatorul Java verifică dacă fiecare referință către variabilă este precedată de execuția unei asignări a variabilei. Dacă nu este așa, atunci se semnalează o eroare la compilare.

Fiecare inițializare (cu excepția primei) este executată dacă precedentă expresie de inițializare se evaluează normal. Execuția unei declarații a unei variabile locale se termină normal dacă evaluarea ultimei expresii de inițializare decurge normal. Evident, dacă declarația unei variabile locale nu conține expresii de inițializare, atunci execuția acesteia se termină întotdeauna normal.

Exemplul 2.7.1. Programul Java de mai jos pune în evidență principiile enunțate anterior:

```

class DeclaratiiVariabileLocale {
    public static void main(String[] args) {
        int a, b, c = (a = 1 + (b = 2)) + 1;
        System.out.println("a = " + a + ", b = " + b + ", c = " + c);
    }
}

```

Compilatorul va produce ieșirea:

```
a = 3, b = 2, c = 4
```

Deoarece o declarație de variabile locale se realizează de la stânga la dreapta, rezultă că se vor declara variabilele locale `a`, `b`, `c` (li se vor aloca în memoria Stack câte

4 octeți la fiecare). Apoi se va evalua expresia de inițializare, care se poate împărți în trei subexpresii. Astfel variabila *b* ia valoarea 2, expresia (*b* = 2) va întoarce valoarea 2, deci variabila *a* va avea valoarea 3. Apoi acestei valori *i* se adaugă 1 și deci valoarea variabilei *c* este 4.

Domeniul unei variabile locale declarate într-un bloc este restul blocului, începând cu instrucțiunea de inițializare. Numele unui parametru variabilă locală sau parametru excepție nu poate fi redeclarat ca variabilă locală, în caz contrar apărând eroare la compilare. Cu alte cuvinte, ascunderea numelui unei variabile locale nu este permisă (cum se întâmplă în cazul variabilelor instanță ale unei clase). O variabilă locală nu poate fi referită folosind un nume calificat (folosind operatorul de acces „.”), ci doar printr-un nume.

Exemplul 2.7.2. Iată un program Java în care punem în evidență domeniul variabilelor:

```
class DeclaratiiVariabileLocale {
    static int x = 6;
    public static void main(String[] args) {
        System.out.println("x = " + x);
        int x = (x = 3) * 4;
        System.out.println("x = " + x);
    }
}
```

Ieșirea programului va fi:

```
x = 6
x = 12
```

La prima afișare este vorba de variabila *x* a clasei *DeclaratiiVariabileLocale* (fiind declarată static, este accesibilă dintr-o metodă statică). La cea de-a doua afișare, este vorba de variabila locală metodei *main()* cu același nume. Domeniul variabilei locale *x* începe imediat după declararea acesteia, deci în expresia (*x* = 3) este vorba de variabila locală, și nu de variabila clasă.

Domeniul unei variabile locale declarate într-o instrucțiune *for* este în toată instrucțiunea, inclusiv inițializatorul. Dacă o declarație a unui identificator ca variabilă locală apare în domeniul unei variabile parametru sau locale cu același nume, atunci apare o eroare la compilare.

Exemplul 2.7.3. Ilustrăm acum marginile domeniilor variabilelor:

```
class DeclaratiiVariabileLocale {
    public static void main(String[] args) {
        {
            int i = 1;
            System.out.println("i = " + i);
        }
    }
}
```

```
for (int i = 0; i < 4; i++)
    System.out.println("i = " + i);
}
```

Ieșirea va fi:

```
i = 1
i = 0
i = 1
i = 2
i = 3
```

Prima apariție a variabilei locale *i* este între cele două acolade (*{*, *}*), iar cea de-a doua apariție este în instrucțiunea *for*. Astfel, domeniile acestor două variabile locale cu același nume sunt disjuncte. Dacă nu ar fi fost prezente cele două acolade, atunci am fi obținut eroare la compilare.

Fiecare declarator dintr-o declarație a unei variabile locale declară o singură variabilă locală, al cărei nume este identificatorul care apare în declarator. Tipul variabilei apare la începutul declarării, urmată eventual de perechi de paranteze pătrate (*[*, *]*) în cazul tablourilor. Tablourile vor fi reluate în Capitolul 4.

Exemplul 2.7.4. Astfel, declarația:

```
int m1, m2[], m3[][];
```

este echivalentă cu următoarele trei declarații:

```
int m1;
int m2[];
int m3[][];
```

Parantezele pătrate se pot pune și la început (asemănător cu limbajele C/C++). De exemplu:

```
int [][] m1, m2[], m3[][];
```

este echivalent cu:

```
int m1[][];
int m2[][][];
int m3[][][][];
```

2.8. Conversii

Fiecare expresie Java are un tip care poate fi determinat din structura expresiei și tipurile literalilor, variabilelor și metodelor menționate în expresie. Sunt cazuri când anumite expresii apar într-un context în care tipul expresiei nu este potrivit, de exemplu, condiția dintr-o instrucțiune *if* trebuie să aibă tipul *boolean* (și nu alt tip),

altfel apare o eroare la compilare. În anumite cazuri, contextul poate fi capabil să accepte un tip apropiat de tipul expresiei. În loc să fie obligatorie scrierea tipului dorit (ceea ce se mai numește *conversie explicită*), Java face o *conversie implicită* a tipului expresiei la un tip acceptabil pentru contextul respectiv.

În fiecare context, sunt permise doar câteva conversii specifice:

- conversii identice;
- conversii primitive implicite („de la mic la mare”, eng. *widening conversions*);
- conversii primitive explicite („de la mare la mic”, eng. *narrowing conversions*);
- conversii de referință implicite;
- conversii de referință explicite;
- conversii la String.

Se numește *conversie identică* o conversie de la un tip la același tip. Astfel, aceasta implică determinarea tipului dorit pentru expresie și, în plus, mărește claritatea programului.

Exemplul 2.8.1. Conversia identică clarifică tipul unui literal. Execuția secvenței de cod Java:

```
System.out.println(32768);
System.out.println((short)32768);
```

va afișa:

```
32768
-32768
```

Explicația este simplă. Prima apariție a literalului 32768 va avea asociat tipul `int`, iar a doua apariție tipul `short`. Cum această din urmă valoare depășește domeniul tipului `short`, va fi afișat `-32768`.

Se numește *conversie primitivă implicită* o conversie de la un tip primitiv cu domeniul mai mic la un tip primitiv cu domeniul mai mare, după cum urmează:

- de la `byte` la `short`, `int`, `long`, `float` și `double`
- de la `short` la `int`, `long`, `float` și `double`
- de la `char` la `int`, `long`, `float` și `double`
- de la `int` la `long`, `float` și `double`
- de la `long` la `float` și `double`
- de la `float` la `double`

Conversiile primitive implicite nu pierd informații despre valoarea numerică inițială, deoarece se adaugă octeți la reprezentarea sa. Totuși, există situații când la conversia dintre un tip integral la un tip în virgulă flotantă se poate pierde precizia, adică se pot pierde ultimii biți ne semnificativi. În acest caz, valoarea în virgulă mobilă se va rotunji la o valoare întreagă apropiată. Justificarea faptului că se poate pierde precizia este foarte simplă: în secțiunea 2.4, am văzut că precizia tipului `float` este 6-7 cifre semnificative, iar a tipului `double` de 14-15 cifre semnificative. Așadar,

considerând un număr întreg de cel puțin 8 cifre, prin conversie implicită la tipul `float` se vor pierde cel puțin 1-2 cifre (Exemplul 2.8.2). În ciuda faptului că se poate pierde precizia, conversiile primitive implicite nu aruncă niciodată excepții.

Exemplul 2.8.2. Următorul program Java pune în evidență faptul că uneori conversia primitivă implicită a două numere întregi mari distincte poate conduce la același număr de tip `float`:

```
class PierdePrecizia {
    public static void main(String[] args) {
        int cinci = 55555555;
        float cinciMobil = cinci;
        System.out.println(cinciMobil);
        int cinciSase = 55555556;
        float cinciSaseMobil = cinciSase;
        System.out.println(cinciSaseMobil);
        // tiparim diferenta dintre acestea
        System.out.println(cinci - (int)cinciMobil);
        System.out.println(cinciSase - (int)cinciSaseMobil);
    }
}
```

Execuția programului de mai sus conduce la afișarea datelor:

```
5.5555558E8
5.5555558E8
-29
-18
```

Se numește *conversie primitivă explicită* o conversie de la un tip primitiv cu domeniul mai mare la un tip primitiv cu domeniu mai mic, după cum urmează:

- de la `byte` la `char`
- de la `short` la `byte` și `char`
- de la `char` la `byte` și `short`
- de la `int` la `byte`, `short` și `char`
- de la `long` la `byte`, `short`, `char` și `int`
- de la `float` la `byte`, `short`, `char`, `int` și `long`
- de la `double` la `byte`, `short`, `char`, `int`, `long` și `float`

Conversiile primitive explicite pot pierde informații despre valoarea numerică inițială, deoarece se elimină octeți semnificativi din reprezentarea sa. Atât la conversia de la tipurile integrale, cât și de la `char`, se păstrează octeți ne semnificativi (cei din dreapta reprezentării în memorie). În plus, poate să difere și semnul valorii rezultat față de semnul valorii inițiale.

Exemplul 2.8.3. Execuția subprogramului Java de mai jos:

```
char c = '\u00EF';
System.out.println((byte)c + " " + (short)c);
```

va afișa:

```
-17 239
```

deoarece reprezentarea în memorie a lui c este:

```
00000000 11101111
```

Așadar, convertind caracterul c la tipul byte se va obține un număr negativ, deoarece bitul de semn al octetului nesemnificativ (al optulea bit de la dreapta la stânga) este 1. Folosind procedeul de calcul al reprezentării numerelor negative din memorie (Exemplul 2.5.13), rezultă că va fi tipărit -17. În ceea ce privește conversia la tipul short, lucrurile sunt mai ușoare, deoarece acesta se reprezintă tot pe doi octeți și astfel va fi afișat 239.

În ceea ce privește conversia explicită a unui număr în virgulă flotantă la un tip integral <tip> (unde <tip> poate fi long, int, char, short, byte), există doi pași:

1. numărul în virgulă flotantă este convertit la long, dacă <tip> este long, sau la int, dacă <tip> este byte, short, char, int, după cum urmează:
 - a) dacă numărul în virgulă flotantă este NaN, rezultatul este 0 (de tip long, respectiv int);
 - b) altfel, dacă numărul în virgulă flotantă nu este infinit, valoarea în virgulă flotantă este rotunjită la o valoare întreagă de tip long sau int, după caz;
 - c) altfel (numărul în virgulă flotantă este infinit), dacă valoarea este prea mică, atunci rezultatul va fi cea mai mică valoare reprezentabilă de tip long sau int, iar dacă valoarea este prea mare, atunci rezultatul va fi cea mai mare valoare reprezentabilă de tip long sau int;
2. dacă <tip> este long sau int, atunci rezultatul conversiei este cel furnizat de primul pas, iar dacă <tip> este char, short, byte, atunci se va returna conversia rezultatului primului pas la <tip>.

Exemplul 2.8.4. Programul Java de mai jos pune în evidență comportarea conversiilor la „capetele” domeniului tipului float:

```
public class ConversiiExpliciteFlotante {
    public static void main(String[] args) {
        float min = Float.NEGATIVE_INFINITY;
        float max = Float.POSITIVE_INFINITY;
        System.out.println("long: " + (long)min + " " + (long)max);
        System.out.println("int: " + (int)min + " " + (int)max);
        System.out.println("char: " + (int)(char)min + " " + (int)(char)max);
        System.out.println("short: " + (short)min + " " + (short)max);
        System.out.println("byte: " + (byte)min + " " + (byte)max);
    }
}
```

La execuție, se va afișa:

```
long: -9223372036854775808 9223372036854775807
int: -2147483648 2147483647
char: 0 65535
short: 0 -1
byte: 0 -1
```

În cazul conversiei la char, am mai realizat o conversie explicită la int pentru a vedea codurile întregi ale caracterelor respective. În ceea ce privește conversiile la tipurile short și byte, acestea nu reprezintă minimul, respectiv maximul domeniilor acestora! După cum s-a precizat în cei doi pași de mai sus, se va face mai întâi conversia la tipul int care va implica obținerea rezultatelor așteptate: 0x80000000 (minimul), respectiv 0x7fffffff. Deci, după trunchierea făcută de conversia pentru short, se vor obține valorile: 0x0000 (adică 0), respectiv 0xffff (adică -1), iar pentru byte se vor obține valorile: 0x00 (adică 0), respectiv 0xff (adică -1).

Conversiile explicite de la double la float folosesc modul de rotunjire a valorii în virgulă flotantă. O valoare prea mică a tipului double este convertită la zero negativ sau pozitiv, după caz; o valoare prea mare a tipului double este convertită la plus/minus infinit, după caz. O valoare NaN de tip double este convertită la valoarea NaN de tip float. În ciuda faptului că conversiile pot pierde informații, acestea nu aruncă excepții în timpul execuției programului.

Exemplul 2.8.5. Execuția programului Java:

```
public class AlteConversiiExpliciteFlotante {
    public static void main(String[] args) {
        System.out.println("Valoare prea mare float: "
            + "(int)1e15f==" + (int)1e15f);
        System.out.println("Float.NaN: (int)NaN=="
            + "(int)Float.NaN);
        System.out.println("Valoare prea mare double: "
            + "(float)-1e40==" + (float)-1e40);
        System.out.println("Valoare prea mica double: "
            + "(float)1e-50==" + (float)1e-50);
    }
}
```

va afișa:

```
Valoare prea mare float: (int)1e15f==2147483647
Float.NaN: (int)NaN==0
Valoare prea mare double: (float)-1e40==Infinity
Valoare prea mica double: (float)1e-50==0.0
```

Conversiile la tipul referință implicite și explicite vor fi prezentate în cadrul Capitolului 3.

2.8.1. Contextele de conversie

În Java, există cinci *contexte de conversie* în care poate apărea conversia expresiilor Java. Un exemplu frecvent de context de conversie este operandul operatorului numeric `+`. Procesul de conversie pentru acești operanzi se numește *conversie numerică la compilare* (eng. *numeric promotion*). De exemplu, `"4"+5` va implica conversia literalului întreg `5` la literalul `"5"` de tip `String`. Astfel, se va apela operatorul `+` pentru doi literali de tip `String`, expresia fiind evaluată la valoarea `"45"`.

Cele cinci tipuri de contexte de conversie sunt:

1. *Conversia de asignare* convertește tipul unei expresii la tipul unei variabile specificate. Aceasta nu aruncă niciodată excepții.
2. *Conversia la apelul unei metode* se aplică fiecărui argument din apelul unei metode sau constructor și, în general, face aceleași conversii pe care le face conversia de asignare. Aceasta nu aruncă niciodată excepții.
3. *Conversia explicită* (eng. *cast*) convertește tipul unei expresii la un tip explicit specificat de operatorul `cast`. Aceasta permite mai multe conversii specifice (cu excepția celui `String`) și anumite conversii explicite la un tip referință pot cauza aruncarea unei excepții la execuție.
4. *Conversia la String* permite oricărui tip să fie convertit la tipul `String`.
5. *Conversia numerică la compilare* (eng. *numeric promotion*) aduce operanzii operatorului numeric la un tip comun pentru efectuarea operației.

Începem cu prezentarea *conversiei de asignare*. Aceasta apare când valoarea unei expresii este asignată unei variabile. Așadar, tipul expresiei trebuie să fie convertit la tipul variabilei. Contextele de asignare permit folosirea conversiei de identitate, conversiei primitive implicite și a conversiei de referință implicite. În plus, o conversie primitivă explicită poate fi folosită, dacă au loc condițiile:

- expresia este o constantă de tip `int`;
- tipul variabilei este `byte`, `short` sau `char`;
- valoarea expresiei este reprezentabilă în tipul variabilei.

Dacă tipul expresiei nu poate fi convertit la tipul variabilei printr-o conversie permisă într-un context de asignare, atunci apare o eroare de compilare. În caz contrar, dacă tipul expresiei poate fi convertit la tipul variabilei printr-o conversie de asignare, atunci se spune că expresia (sau valoarea sa) este *asignabilă* variabilei (sau, echivalent, tipul expresiei este *compatibil la asignare* cu tipul variabilei). O conversie de asignare nu aruncă niciodată excepții (cu excepția situației când sunt implicate elementele unui tablou). O valoare a unui tip primitiv nu poate fi asignată unei variabile de tip referință, în caz contrar se va obține o eroare de compilare (contextele de asignare ale tipului referință vor fi prezentate în Capitolul 3). O valoare de tip `boolean` poate fi asignată doar unei variabile de tip `boolean`.

Exemplul 2.8.6. Următorul exemplu pune în evidență mai multe situații de contexte de asignare:

```
class ContextAsignare {
    public static void main(String[] args) {
```

```
// literalul 34 de tip int este convertit la asignare
// la tipul short
short s = 34;
// variabila s este convertita la asignare la tipul float
float f = s;
System.out.println("f = " + f);
char c = '\u0123';
// variabila l este convertita la asignare la tipul long
long l = c;
System.out.println("l = " + l);
// f = 34.0
// variabila f este convertita la asignare la tipul float
double d = f;
System.out.println("d = " + d);
}
```

Acesta va afișa la execuție:

```
f = 34.0
l = 291
d = 4.550000190734863
```

Continuăm cu *conversia la apelul unei metode*. Aceasta se aplică fiecărei valori a argumentelor din apelul unei metode sau constructor. Tipul expresiei argument trebuie să poată fi convertit la tipul parametrului corespunzător. Contextul apelului unei metode permite utilizarea conversiei de identitate, a conversiei primitive și de referință implicite.

Exemplul 2.8.7. Următorul subprogram Java pune în evidență folosirea conversiei la apel:

```
float f = 35.7f;
double d = Math.sin(f);
System.out.println("Math.sin(" + f + ") = " + d);
```

Metoda statică `sin()` din clasa `Math` are argumentul de tip `double`, deci în momentul apelului său se va face o conversie a lui `f` la tipul `double`.

Conversiile la apelul unei metode nu includ conversia implicită a constantelor întregi, cum era cazul la conversiile de asignare.

Exemplul 2.8.8. Compilarea programului Java de mai jos:

```
public class ConversieApelMetoda {
    static int f(byte a, int b) { return a + b; }
    static int f(short a, short b) { return a + b; }
    public static void main(String[] args) {
        System.out.println(f(3, 7));
    }
}
```

va conduce la o eroare de compilare. Aceasta, deoarece literalii întregi 3 și 7 au tipul `int` și nu găsim nici o metodă `f()` care să aibă ambii parametri de tip `int`. De exemplu, putem rezolva apelul prin conversie explicită:

```
System.out.println(f((short)3, (short)7));
```

sau:

```
System.out.println(f((byte)3, 7));
```

Continuăm cu *conversiile explicite*. Acestea se aplică operandului operatorului `cast`, precedând o valoare cu un nume de tip între paranteze. Astfel, tipul expresiei operandului trebuie convertit la tipul numit explicit de operatorul `cast`. Contextul conversiilor explicite permite folosirea conversiei de identitate, conversiilor primitive și de referință implicite și explicite. Anumite conversii explicite conduc la erori de compilare. O valoare a unui tip primitiv poate fi convertită la tipul unei conversii de identitate, la tipul unei conversii primitive implicite sau explicite. O valoare a unui tip primitiv nu poate fi convertită la un tip referință prin conversie explicită și nici invers, adică o valoare a unui tip referință nu poate fi convertită explicit la un tip primitiv. Conversiile explicite dintre tipurile referință se vor prezenta în Capitolul 3.

Exemplul 2.8.9. Următoarea linie de cod Java:

```
int i = 3.46f;
```

conduce la o eroare la compilare, deoarece un literal în virgulă flotantă nu se poate converti implicit la tipul `int`. Trebuie utilizată o conversie explicită, astfel:

```
int i = (int)3.46f;
```

Continuăm cu prezentarea *conversiei la String*. Există conversii la tipul `String` ale oricărui alt tip, inclusiv tipul `null`. Conversia la `String` se aplică operanzilor operatorului binar `+` când unul din argumente este `String`. În acest caz special, celălalt argument este convertit la `String` creându-se un nou `String` care este concatenarea celor două `String`-uri (Exemplul 2.5.19).

Conversia numerică la compilare se aplică operanzilor unui operator aritmetic. Contextul conversiei numerice la compilare permite utilizarea conversiei de identitate sau a conversiei primitive implicite. Conversiile numerice la compilare sunt utile la conversia operanzilor unui operator numeric la un tip comun astfel încât operația să se poată efectua. Există două tipuri de conversii numerice la compilare: *unare* și *binare* (după cum conversia se aplică unui operand sau ambilor operanzi).

În cazul *conversiei numerice unare la compilare*, expresiile operatorilor unari întorc o valoare de tip numeric, după cum urmează:

1. Dacă operandul este de tip `byte`, `short`, `char`, atunci acesta se convertește la o valoare de tip `int` prin conversie implicită;
2. În caz contrar, operandul rămâne așa cum este (adică nu i se aplică nici o conversie).

Conversia numerică unară la compilare se execută asupra expresiilor în următoarele situații:

1. în expresia de dimensiune a creării unui tablou;
2. în expresia de index a accesului la elementele unui tablou;
3. în cazul operanzilor operatorilor unari `+` și `-`;
4. în cazul operandului operatorului complement față de doi;
5. fiecărui operand al operatorilor de deplasare `<<`, `>>`, `>>>` (chiar dacă operandul din dreapta este de tip `long`, acest lucru nu va implica și conversia primului operand la `long`).

Pentru comportarea operatorilor de deplasare se poate consulta Exemplul 2.5.11.

Exemplul 2.8.10. Următorul program Java pune în evidență toate categoriile de conversii implicite unare la compilare (vom reveni asupra celor referitoare la tablouri în Capitolul 4):

```
public class ConversiiImpliciteUnareLaCompilare {
    public static void main(String[] args) {
        byte b = 2;
        // urmeaza o conversie la compilare pentru dimensiunea
        // unui tablou
        int v[] = new int[b];
        // caracterul c va avea codul 1
        char c = '\u0001';
        // urmeaza o conversie la compilare pentru indexul unui
        // tablou
        v[c] = 1;
        // urmeaza o conversie la compilare pentru minus unar
        v[0] = -c;
        System.out.println("v[0] = " + v[0] + ", v[1] = " + v[1]);
        // asignam b cu -1. Reprezentarea acestuia in memorie
        // este cu toti bitii 1
        b = -1;
        // urmeaza o conversie la compilare pentru operatorul
        // complement fata de doi
        int i = ~b;
        System.out.println("~0x" + Integer.toBinaryString(b)
            + " = 0x" + Integer.toBinaryString(i));
        // urmeaza o conversie la compilare pentru operatorul
        // de deplasare stanga
        i = b << 4L;
        System.out.println("0x" + Integer.toBinaryString(b)
            + "<<4L = 0x" + Integer.toBinaryString(i));
    }
}
```

Acesta va afișa la execuție:

```
v[0] = -1, v[1] = 1
~0x11111111111111111111111111111111 = 0x0
0x11111111111111111111111111111111 < 4L =
0x111111111111111111111111111111110000
```

Conversia numerică binară la compilare se aplică unei perechi de operanzi, care trebuie să noteze un tip numeric și care trebuie să îndeplinească următoarele reguli de conversie implicită a operanzilor:

1. Dacă unul dintre operanzi este de tip `double`, atunci și celălalt va fi convertit la `double`;
2. Altfel, dacă unul dintre operanzi este de tip `float`, atunci și celălalt va fi convertit la `float`;
3. Altfel, dacă unul din operanzi este de tip `long`, atunci și celălalt va fi convertit la `long`;
4. Altfel, ambii operanzi vor fi converțiți la `int`.

Conversia numerică binară la compilare se execută asupra expresiilor în următoarele situații:

1. Pentru operatorii multiplicativi `*`, `/` și `%`
2. Pentru operatorii de adunare și scădere `+` și `-` pentru tipurile numerice
3. Pentru operatorii de comparație numerici `<`, `<=`, `>`, `>=`
4. Pentru operatorii de egalitate numerici `==` și `!=`
5. Pentru operatorii pe biți care operează pe tipuri integrale `&`, `^` și `|`
6. Pentru operatorul condițional `?` : (în anumite cazuri)

Exemplul 2.8.11. Următorul program Java pune în evidență câteva din categoriile de conversii implicite binare la compilare:

```
public class ConversiiImpliciteBinareLaCompilare {
    public static void main(String[] args) {
        int i = 0;
        float f = 1.0f;
        double d = 2.0;
        // Produsul i * f este convertit la compilare
        // in float * float, apoi testul
        // float == double este convertit
        // in double == double
        if (i * f == d)
            System.out.println("Are loc egalitatea.");
        // O expresie char & byte este convertita la int & int
        byte b = 31; // b are reprezentarea 00011111
        char c = 'G'; // c are reprezentarea 00000000 01000111
        int cSiB = c & b;
```

```
System.out.println(Integer.toBinaryString(cSiB));
// O expresie a operatorului conditional
// boolean ? int : float este convertita
// la compilare boolean ? float : float
System.out.println((1 < 2) ? i : 4.0f);
}
```

Acesta va afișa la execuție:

```
111
0.0
```

La afișarea reprezentării binare a valorii variabilei `cSiB`, s-au eliminat zerourile semnificative (în număr de 29). În ceea ce privește operatorul condițional `?` :, mai întâi se convertește variabila `i` la tipul `float`, apoi se întoarce valoarea `0.0` de tip `float` (care se afișează).

2.9. Structuri de control

Execuția unui program Java este controlată de *structuri de control* sau *instrucțiuni*, care sunt executate pentru efectul lor și nu pentru faptul că întorc valori (cum era cazul expresiilor, Secțiunea 2.5). Anumite instrucțiuni conțin alte instrucțiuni (sau expresii) ca parte a structurii lor. Spunem că instrucțiunea I_1 conține (imediat) instrucțiunea I_2 dacă nu există nici o instrucțiune I_3 diferită de I_1 și I_2 astfel încât I_1 conțină I_3 și I_3 conțină I_2 . Instrucțiunile Java sunt: instrucțiunea vidă, etichetele, expresiile, `if`, `switch`, `while`, `do`, `for`, `break`, `continue`, `return`, `throw`, `synchronized`, `try`. Spre deosebire de alte limbaje de programare, în Java nu există instrucțiunea `goto`. Totuși cuvântul `goto` este rezervat.

Fiecare instrucțiune are un mod normal de execuție în care sunt făcuți anumiți pași de calcul. Dacă toți pașii au loc fără vreo terminare anormală, atunci spunem că instrucțiunea s-a terminat normal. Cauzele posibile ale terminării anormale ale unei instrucțiuni sunt:

1. instrucțiunile `break`, `continue` și `return` pot cauza transferul controlului programului prin terminarea anormală a unei instrucțiuni care le conține;
2. evaluarea anumitor expresii Java sau instrucțiunile `throw` pot arunca excepții ale mașinii virtuale Java. O excepție poate cauza transferul controlului programului prin terminarea anormală a unei instrucțiuni care o conține.

Motivele pentru care poate apărea o terminare anormală a unei instrucțiuni sunt: `break` (cu sau fără etichetă), `continue` (cu sau fără etichetă), `return` (cu sau fără valoare precizată) și `throw` (cu valoare precizată, incluzând excepțiile aruncate de mașina virtuală Java). Dacă o instrucțiune evaluează o expresie, terminarea anormală a expresiei va implica și terminarea anormală a întregii instrucțiuni (din același motiv). Toți pașii care ar fi urmat în execuție nu mai sunt făcuți.

Înainte de a începe prezentarea sintaxei și semanticii instrucțiunilor Java, vom descrie noțiunea de *bloc* (de instrucțiuni). Un *bloc* (de instrucțiuni) este o secvență de instrucțiuni (pot fi și instrucțiuni de declarare a variabilelor locale) cuprinsă între acolade. Un bloc este executat prin execuția fiecărei instrucțiuni a blocului de la prima la ultima și de la stânga la dreapta.

Declarațiile variabilelor locale s-au prezentat în Secțiunea 2.7. O declarație de variabilă locală este o instrucțiune executabilă. La execuția acesteia, declaratorii de variabile sunt procesați de la stânga la dreapta. Dacă un declarator are o expresie de inițializare, expresia este evaluată și valoarea este asignată variabilei. Dacă declaratorul unei variabile (să o notăm cu *v*, de exemplu) nu are o expresie de inițializare, atunci compilatorul Java va verifica dacă fiecare referire la variabila *v* este precedată de o inițializare a acesteia. În caz contrar, se va obține o eroare încă de la compilare, primind un mesaj de genul: „variable *v* might not have been initialized”. Fiecare inițializare (cu excepția primei) se execută doar dacă evaluarea expresiilor de inițializare precedente se termină normal. Dacă declarația de variabile locale nu conține expresii de inițializare, atunci execuția acesteia se termină mereu cu succes.

2.9.1. Instrucțiunea vidă

Instrucțiunea vidă are sintaxa `;` și nu face nimic. Aceasta se termină normal mereu.

2.9.2. Instrucțiunea etichetă

Instrucțiunea etichetă constă din numele unui identificator, urmat de caracterul două puncte („:”). După caracterul „:” urmează o instrucțiune Java. Instrucțiunile etichetă sunt utilizate împreună cu instrucțiunile `break` și `continue`. O instrucțiune etichetă nu poate apărea în altă instrucțiune etichetă cu același identificator, în caz contrar apare o eroare la compilare. Două instrucțiuni etichetă pot avea același identificator doar dacă nu sunt incluse una în alta.

Nu există restricții asupra utilizării aceluiași identificator ca etichetă și ca nume de pachet, clasă, interfață, metodă, atribut, parametru sau variabilă locală.

O instrucțiune etichetă se execută prin execuția instrucțiunii care urmează după „:”. Dacă instrucțiunea este etichetată de un identificator `id` și instrucțiunea care urmează după „:”, notată cu `Instr`, se termină anormal din cauza unui `break` cu același `id`, atunci spunem că instrucțiunea de etichetă `id` s-a terminat normal. În toate celelalte cazuri de terminare anormală a lui `Instr`, spunem că instrucțiunea de etichetă `id` s-a terminat anormal.

Exemplul 2.9.1. Următorul program Java pune în evidență modalitatea de utilizare a instrucțiunilor etichetă (precum și a instrucțiunilor `for`, `break` etc. care se vor prezenta ulterior):

```
public class InstrucțiuneEticheta {
    public static void main(String args[]) {
```

```
        int max = 5;
        for (int i = 0; i < max; i++) {
            unu: {i--;
                System.out.println("i (unu) = " + i);
                for (int j = 0; j < max; j++)
                    if (j == i) break unu;
                System.out.println("S-a executat instructiunea"
                    + "for (unu)");
            }
            doi: {i += 2;
                System.out.println("i (doi) = " + i);
                for (int j = 0; j < max; j++)
                    if (j == i) break doi;
                System.out.println("S-a executat instructiunea"
                    + "for (doi)");
            }
            System.out.println("i = " + i);
        }
    }
}
```

La execuție se va afișa:

```
i (unu) = -1
S-a executat instructiunea for (unu)
i (doi) = 1
i = 1
i (unu) = 1
i (doi) = 3
i = 3
i (unu) = 3
i (doi) = 5
S-a executat instructiunea for (doi)
i = 5
```

La prima iterație a instrucțiunii `for`, valoarea lui `i` este `-1`, deci se va executa instrucțiunea `for` interioară fără a se ieși necondiționat. Apoi, se va executa instrucțiunea etichetată cu `doi`, care va implica ieșirea din instrucțiunea `for` interioară când valoarea lui `j` este `1`. După cele trei iterații, se va termina execuția întregului program.

2.9.3. Instrucțiunea expresie

Instrucțiunea expresie are forma generală `<expresie>;`, unde `<expresie>` poate fi o asignare, expresie de preincrementare, predecrementare, postincrementare, postdecrementare, apelul unei metode, expresie de creare a unei instanțe a unei clase. O instrucțiune expresie se execută prin evaluarea sa (adică a lui `<expresie>`).

Dacă aceasta are o valoare, atunci aceasta este returnată. Execuția unei instrucțiuni expresie se termină normal dacă și numai dacă evaluarea expresiei se termină normal.

În Java, nu există conversie către void (cum era în alte limbaje de programare: C, C++). Aceasta se poate simula prin instrucțiuni de asignare sau prin instrucțiuni de declarare a unei variabile locale.

Instrucțiunile expresie sunt utilizate frecvent, cum ar fi în Exemplele 2.5.1, 2.5.2, 2.5.3, 2.5.9, 2.5.25, 2.7.1, 2.7.2.

2.9.4. Instrucțiunea if

Instrucțiunea *if* are două forme: simplă (eng. *if-then*) și completă (eng. *if-then-else*). Instrucțiunea *if* permite execuția condiționată a unei instrucțiuni. Sintaxa lui *if-then* este:

```
if <Expresie> <Instrucțiune>
```

iar a lui *if-then-else* este:

```
if <Expresie> <Instrucțiune1> else <Instrucțiune2>
```

unde <Expresie> trebuie să aibă tipul boolean, în caz contrar se obține o eroare de compilare.

În cazul instrucțiunii *if-then* se evaluează <Expresie>. Dacă evaluarea se termină anormal, atunci instrucțiunea *if* se termină anormal. Dacă evaluarea lui <Expresie> este true, atunci se execută <Instrucțiune> și astfel instrucțiunea *if* se termină normal. Dacă evaluarea lui <Expresie> este false, atunci nu se execută nimic și instrucțiunea *if* se termină normal.

În cazul instrucțiunii *if-then-else* se evaluează <Expresie>. Dacă evaluarea se termină anormal, atunci instrucțiunea *if* se termină anormal. Dacă evaluarea lui <Expresie> este true, atunci se execută <Instrucțiune1> și astfel instrucțiunea *if* se termină normal. Dacă evaluarea lui <Expresie> este false, atunci se execută <Instrucțiune2> și instrucțiunea *if* se termină normal.

O situație deosebită în cazul instrucțiunii *if* o reprezintă asocierea ramurii *else* în mod corect. Pentru claritate, se pot utiliza acolade pentru a delimita blocul în care este scrisă instrucțiunea *if*.

Exemplul 2.9.2. Subprogramul Java:

```
int a = 3;
if (++a < 4)
    System.out.println(a);
else
    System.out.println(a);
```

nu va afișa nimic la execuție întrucât ramura *else* este asociată ultimului *if*. În schimb, dacă adăugăm pentru delimitarea blocului primei instrucțiuni *if* două acolade, atunci obținem codul:

```
int a = 3;
if (++a < 4) {
```

```
    if (++a < 4)
        System.out.println(a);
    }
    else
        System.out.println(a);
```

care va afișa la execuție valoarea 4.

2.9.5. Instrucțiunea switch

Instrucțiunea *switch* transferă controlul uneia din mai multe instrucțiuni depinzând de valoarea expresiei. Sintaxa generală este:

```
switch (<Expresie>) {
    default: <Instrucțiune0>
    case <ExpresieConstanta1> : <Instrucțiune1>
    case <ExpresieConstanta2> : <Instrucțiune2>
    . . .
    case <ExpresieConstantaN> : <InstrucțiuneN>
}
```

Ordinea în care apar clauzele *case* nu are importanță, iar clauza *default* poate fi scrisă oriunde. Tipul lui <Expresie> trebuie să fie char, byte, short sau int, în caz contrar apare o eroare de compilare. Corpul instrucțiunii *switch* trebuie să fie un bloc. Orice instrucțiune conținută de bloc poate fi etichetată de una sau mai multe etichete *case* și *default*. Aceste etichete se spune că sunt asociate instrucțiunii *switch*, ca valori ale expresiilor constante din etichetele *case*.

În cadrul unei instrucțiuni *switch*, următoarele cerințe trebuie satisfăcute, altfel obținem o eroare la compilare:

1. Fiecare valoare a lui <ExpresieConstanta1>, <ExpresieConstanta2>, ..., <ExpresieConstantaN> trebuie să fie asignabilă cu tipul lui <Expresie>;
2. Nu trebuie să existe două expresii constante care să aibă aceeași valoare;
3. Cel mult o etichetă *default* poate fi asociată cu aceeași instrucțiune *switch*.

Când este executată o instrucțiune *switch*, mai întâi se evaluează <Expresie>. Dacă evaluarea lui <Expresie> se termină anormal, atunci instrucțiunea *switch* se termină anormal. Altfel, execuția continuă prin compararea valorii lui <Expresie> cu fiecare expresie *case*. Atunci:

1. Dacă una din constantele *case* (adică <ExpresieConstantaK>, unde K este 1, 2, ..., N) este egală cu valoarea lui <Expresie>, atunci se execută toate instrucțiunile după eticheta *case* până la întâlnirea primului *break*. Dacă toate aceste instrucțiuni se execută normal, atunci întreaga instrucțiune *switch* se termină normal.
2. Dacă nu există nici o etichetă care să se potrivească cu valoarea lui <Expresie> și există o etichetă *default*, atunci se execută toate instrucțiunile după eticheta

default până la întâlnirea primului break. Dacă toate aceste instrucțiuni se execută normal sau nu există nici o instrucțiune după eticheta default, atunci întreaga instrucțiune switch se termină normal.

3. Dacă nu există nici o etichetă case care să se potrivească cu valoarea lui <Expresie> și nu există o etichetă default, atunci nu se execută nimic și instrucțiunea switch se termină normal.

Dacă execuția unei <InstrucțiuniK>, unde K este 1, 2, ..., N, se termină anormal din cauza unei instrucțiuni break fără etichetă, atunci nu se execută nimic și instrucțiunea switch se termină normal. Dacă execuția unei <InstrucțiuniK> se termină anormal din alt motiv, atunci instrucțiunea switch se termină anormal din același motiv. Cazul terminării anormale din cauza unei instrucțiuni break cu etichetă se rezolvă folosind regula generală pentru instrucțiuni etichetate.

Exemplul 2.9.3. Următorul program Java scoate în evidență folosirea instrucțiunii switch:

```
public class Switch {
    static void f(int k) {
        switch (k) {
            default: System.out.println("implicit"); break;
            case 1: System.out.println("unu"); break;
            case 2: case 3: System.out.println("doi sau trei"); break;
            case 4: case 5: System.out.println("patru sau cinci");
        }
    }

    public static void main(String[] args) {
        f(1);
        f(2);
        f(3);
        f(4);
        f(6);
    }
}
```

La execuție, acesta va afișa:

```
unu
doi sau trei
doi sau trei
patru sau cinci
implicit
```

Se observă posibilitatea utilizării mai multor expresii case asociate unui același bloc de instrucțiuni. Este esențială și terminarea unui caz cu ajutorul instrucțiunii

break. Dacă, de exemplu, n-ar fi fost primul break asociat cazului default, atunci rezultatul ar fi fost:

```
unu
doi sau trei
doi sau trei
patru sau cinci
implicit
unu
```

2.9.5. Instrucțiunea while

Instrucțiunea while este întrebuințată pentru execuția repetată a unei instrucțiuni și are sintaxa generală:

```
while (<Expresie>) <Instrucțiune>
```

unde <Expresie> trebuie să aibă tipul boolean, altfel obținem o eroare de compilare. Execuția instrucțiunii while începe prin evaluarea lui <Expresie>. Dacă evaluarea lui <Expresie> se termină anormal, atunci și instrucțiunea while se termină anormal. Altfel, execuția continuă astfel:

1. Dacă valoarea lui <Expresie> este true, atunci se execută <Instrucțiune>.
 - a) Dacă execuția lui <Instrucțiune> se termină normal, atunci întreaga instrucțiune while se execută din nou, începând cu reevaluarea lui <Expresie>.
 - b) Dacă execuția lui <Instrucțiune> se termină anormal, atunci se execută pasul 3.
2. Dacă valoarea lui <Expresie> este false, atunci nu se execută nimic și execuția instrucțiunii while se termină normal. De aceea, dacă valoarea lui <Expresie> este false pentru prima evaluare (iterație), atunci <Instrucțiune> nu se execută.
3. Dacă execuția lui <Instrucțiune> se termină anormal din cauza unui break fără etichetă, atunci nu se execută nimic și execuția instrucțiunii while se termină normal.
4. Dacă execuția lui <Instrucțiune> se termină anormal din cauza unui continue fără etichetă, atunci instrucțiunea while se execută încă o dată.
5. Dacă execuția lui <Instrucțiune> se termină anormal din cauza unui continue cu eticheta E, atunci:
 - a) Dacă instrucțiunea while are eticheta E, atunci întreaga instrucțiune while se execută încă o dată;
 - b) Dacă instrucțiunea while nu are eticheta E, atunci întreaga instrucțiune while se termină anormal;
6. Dacă execuția lui <Instrucțiune> se termină anormal din cauza unui alt motiv, atunci întreaga instrucțiune while se termină anormal din același motiv. Cazul terminării anormale datorat unui break cu etichetă se tratează după regulile instrucțiunilor cu etichetă.

Astfel, o instrucțiune while se poate executa de zero ori (dacă <Expresie> este evaluată la false încă de la început), de un număr finit de ori (dacă <Expresie> se evaluează la true sau există un break în <Instrucțiune>; acesta este cazul cel mai des întâlnit) sau de o infinitate de ori (dacă <Instrucțiune> nu reușește să implice evaluarea lui <Expresie> la true sau nu există un break în <Instrucțiune>; acest caz implică intrarea execuției programului într-o „bucă infinite”). În general, se recomandă utilizarea lui while când nu se cunoaște aprioric numărul de iterații.

Exemplul 2.9.4. Iată un prim exemplu de utilizare a instrucțiunii while care se termină cu ajutorul instrucțiunii break:

```
public class WhileUnu {
    public static void main(String args[]) {
        int i = 0;
        while (true) {
            if (i == 1) break;
            i = (int) (Math.random() * 5);
            System.out.println("i = " + i);
        }
    }
}
```

Acesta va afișa (eventual) la execuție:

```
i = 2
i = 3
...
i = 1
```

Execuția programului de mai sus este nedeterministă, adică la două execuții distincte este foarte posibil să obținem rezultate diferite.

Exemplul 2.9.5. Un al doilea exemplu de utilizare a instrucțiunii while condiționat (adică de ieșire cu satisfacerea condiției booleene, și nu cu break) este:

```
public class WhileDoi {
    public static int f(int n) {
        return (int) (Math.random() * n);
    }

    public static void main(String args[]) {
        int i = 0;
        while (f(5) != 1) i++;
        System.out.println("au fost " + i + " iteratii.");
    }
}
```

O posibilă execuție a acestui program este:

au fost 7 iteratii.

2.9.6. Instrucțiunea do

Ca și while, instrucțiunea do face parte din categoria instrucțiunilor repetitive și implică execuția cel puțin o dată a unei instrucțiuni. Sintaxa generală este:

```
do
    <Instrucțiune>
while (<Expresie>);
```

Tipul lui <Expresie> trebuie să fie boolean, altfel se obține eroare la compilare. Semantica instrucțiunii do este simplă. Astfel, se execută mai întâi <Instrucțiune>:

1. Dacă execuția lui <Instrucțiune> se termină normal, atunci se evaluează <Expresie>. Dacă evaluarea lui <Expresie> se termină anormal, atunci și instrucțiunea do se termină anormal. Altfel, avem:
 - a) Dacă valoarea lui <Expresie> este true, se execută din nou instrucțiunea do;
 - b) Dacă valoarea lui <Expresie> este false, nu se execută nimic și instrucțiunea do se termină normal.
2. Dacă execuția lui <Instrucțiune> se termină anormal, atunci:
 - a) Dacă execuția lui <Instrucțiune> se termină anormal din cauza unui break fără etichete, atunci nu se execută nimic și instrucțiunea do se termină normal;
 - b) Dacă execuția lui <Instrucțiune> se termină anormal din cauza unui continue fără etichete, atunci se evaluează <Expresie> și avem cazurile 1. a) și respectiv 1. b);
 - c) Dacă execuția lui <Instrucțiune> se termină anormal din cauza unei etichete E, atunci avem:
 - i. Dacă instrucțiunea do are eticheta E, atunci se evaluează <Expresie> și avem cazurile 1. a) și respectiv 1. b);
 - ii. Dacă instrucțiunea do nu are eticheta E, atunci instrucțiunea do se termină anormal.
3. Dacă execuția lui <Instrucțiune> se termină anormal din alt motiv, atunci instrucțiunea do se termină anormal din același motiv. Cazul terminării anormale datorat unui break cu etichetă se tratează după regulile instrucțiunilor cu etichetă.

Exemplul 2.9.6. Se pot reformula Exemplele 2.9.4 și 2.9.5 astfel încât să simuleze iterațiile cu instrucțiunea do, și nu cu while. În schimb, vom prezenta un exemplu puțin mai complicat care citește un număr arbitrar de numere în virgulă flotantă de la tastatură, apoi afișează suma acestora. Vom vedea că în cazul appleturilor, citirea numerelor de la tastatură se face de către bucla evenimentelor. Programul conține și

noțiuni de fișiere necesare citirii șirurilor de caractere de la tastatură. Iată un program Java de sine stătător care implementează această problemă:

```
import java.io.*;

public class DoWhile {
    public static void main(String[] args) {
        BufferedReader tastatura = new
            BufferedReader(new InputStreamReader(System.in),1);
        double suma = 0.0, numarCitit = 0.0;
        String linie = "";
        do {
            try {
                System.out.flush();
                System.out.println("Dati un numar in virgula flotanta");
                linie = tastatura.readLine();
                Double tempDouble = Double.valueOf(linie);
                numarCitit = tempDouble.doubleValue();
                System.out.println("numarCitit = " + numarCitit);
                suma += numarCitit;
                System.out.println("Doriti sa continuati (d/n)?");
                System.out.flush();
                linie = tastatura.readLine();
                System.out.println("Ati tastat " + linie);
            }
            catch (IOException e) {
                System.out.println("Intrare de la tastatura " +
                    e.toString());
                System.exit(1);
            }
        }
        while (linie.equals("d") || linie.equals("D"));
        System.out.println("Suma numerelor citite este " + suma);
        try {
            tastatura.close();
        }
        catch (IOException e) {
            System.out.println("Eroare inchidere fisier de intrare "
                + e.toString());
            System.exit(2);
        }
    } // sfarsitul metodei main()
} // sfarsitul definitiei clasei DoWhile
```

Iată o posibilă execuție a acestui program:

```
Dati un numar in virgula flotanta
2.98
numarCitit = 2.98
Doriti sa continuati (d/n)?
d
Ati tastat d
Dati un numar in virgula flotanta
4.78
numarCitit = 4.78
Doriti sa continuati (d/n)?
d
Ati tastat d
Dati un numar in virgula flotanta
-3.12
numarCitit = -3.12
Doriti sa continuati (d/n)?
n
Ati tastat n
Suma numerelor citite este 4.64
```

2.9.7. Instrucțiunea for

Ultima instrucțiune în categoria celor repetitive este for. În general, aceasta este utilă când se cunoaște numărul (maxim) de iterații care vor trebui efectuate. Sintaxa generală este:

```
for ([<ListaExpresieInitializare>; [<Expresie>] ;
    [<ListaExpresieActualizare>])
    <Instrucțiune>
```

După cum se poate observa, în instrucțiunea for toate cele trei expresii sunt optionale. <ListaExpresieInitializare> și <ListaExpresieActualizare> reprezintă liste de expresii și instrucțiuni separate prin virgulă. În plus, <ListaExpresieInitializare> poate conține și expresii de declarare a variabilelor locale. Tipul lui <Expresie> trebuie să fie boolean, altfel obținem o eroare de compilare.

Iată cum se poate scrie instrucțiunea for de mai sus utilizând o instrucțiune while și păstrând echivalența semantică:

```
<ListaExpresieInitializare>;
while (<Expresie>) {
    <Instrucțiune>
    <ListaExpresieActualizare>;
}
```

Să vedem exact modul de execuție a instrucțiunii `for` de mai sus începând cu `<ListaExpresieInitializare>`:

1. Dacă `<ListaExpresieInitializare>` este o listă de expresii instrucțiune, atunci expresiile sunt evaluate de la stânga la dreapta. Dacă vreo expresie se evaluează anormal, atunci întreaga instrucțiune `for` se termină anormal.
2. Dacă `<ListaExpresieInitializare>` este o declarație de variabilă locală, aceasta este executată ca și cum ar fi fost instrucțiune de declarație a unei variabile locale într-un bloc. În acest caz, domeniul variabilei locale declarate este în `<ListaExpresieInitializare>`, `<Expresie>`, `<ListaExpresieActualizare>` și `<Instrucțiune>`. Dacă execuția declarației variabilei locale se termină anormal dintr-un anumit motiv, atunci instrucțiunea `for` se termină anormal din același motiv.
3. Dacă `<ListaExpresieInitializare>` nu este prezent, atunci nu se execută nimic.

O iterație a instrucțiunii `for` se execută astfel:

1. Dacă `<Expresie>` este prezentă, atunci aceasta se evaluează. Dacă `<Expresie>` se evaluează anormal, atunci întreaga instrucțiune `for` se termină anormal. Altfel:
 - a) Dacă `<Expresie>` se evaluează la `true`, atunci se execută `<Instrucțiune>`.
 - b) Dacă `<Expresie>` este evaluat la `false`, atunci nu se mai execută nimic și instrucțiunea `for` se termină normal. Dacă `<Expresie>` se evaluează la `false` încă de la prima iterație, atunci nu se execută `<Instrucțiune>`.
2. Dacă `<Expresie>` nu este prezentă, atunci singura posibilitate de a termina execuția instrucțiunii `for` este întâlnirea unei instrucțiuni `break`.

La execuția lui `<Instrucțiune>` distingem următoarele cazuri:

1. Dacă execuția lui `<Instrucțiune>` se termină anormal din cauza unui `break` fără etichetă, atunci nu se mai execută nimic, instrucțiunea `for` se termină normal.
2. Dacă execuția lui `<Instrucțiune>` se termină anormal din cauza unui `continue` fără etichetă, atunci în cazul în care există `<ListaExpresieActualizare>`, expresiile sunt evaluate de la stânga la dreapta, iar dacă vreo expresie se evaluează anormal, atunci instrucțiunea `for` se termină anormal. Dacă nu există `<ListaExpresieActualizare>`, atunci nu se execută nimic. Apoi, se trece la următoarea iterație a lui `for`.
3. Dacă execuția lui `<Instrucțiune>` se termină anormal din cauza unui `continue` cu eticheta `E`, în cazul în care instrucțiunea `for` nu are eticheta `E`, atunci instrucțiunea `for` se va termina anormal, iar dacă are eticheta `E`, se continuă cu cele două etape de la pasul 5.
4. Dacă execuția lui `<Instrucțiune>` se termină anormal din anumite motive, atunci instrucțiunea `for` se termină anormal din același motiv. Cazul terminării unui `break` cu etichetă se face la fel ca la instrucțiunile etichetă.
5. Dacă execuția lui `<Instrucțiune>` se termină normal, atunci se parcurg următoarele etape:
 - dacă există `<ListaExpresieActualizare>`, atunci expresiile sunt evaluate de la stânga la dreapta (în cazul în care vreo expresie se evaluează anormal,

- atunci instrucțiunea `for` se termină anormal); dacă nu există `<ListaExpresieActualizare>`, atunci nu se execută nimic;
- se trece la următoarea iterație a instrucțiunii `for`.

Exemplul 2.9.7. Considerăm o aplicație simplă pentru calculul sumei numerelor de la 1 la `n`. Presupunem că `i`, `n` și `suma` sunt trei variabile de tip `int` declarate mai sus. Variabila `n` are deja o valoare, de preferat mai mare decât 1. O primă variantă ar fi:

```
for (suma = 0, i = 1; i <= n; ++i)
    suma += i;
```

Aceasta se poate scrie și mai compact, astfel:

```
for (suma = 0, i = 1; i <= n; suma += i, ++i);
```

Atenție însă la ordinea de scriere dintre `suma += i` și `++i`. Inversând cele două instrucțiuni, se obține un alt rezultat pentru `suma` (s-ar obține valoarea $2+3+\dots+(n+1)$ în loc de $1+2+\dots+n$).

Cum s-a menționat și în semantica instrucțiunii `for` de mai sus, cele trei părți ale lui `for` pot lipsi (fiecare sau cumulativ). Astfel, putem elimina `<ListaExpresieInitializare>` și putem scrie un cod echivalent:

```
i = 1;
suma = 0;
for ( ; i <= n; ++i)
    suma += i;
```

Putem elimina și `<ListaExpresieActualizare>`, obținând un cod echivalent:

```
i = 1;
suma = 0;
for ( ; i <= n; )
    suma += i++;
```

Dacă, în schimb, lipsește și `<Expresie>`, trebuie să utilizăm instrucțiunea `break`, în caz contrar obținem o buclă infinită.

```
i = 1;
suma = 0;
for ( ; ; ) {
    suma += i++;
    if (i > n) break;
}
```

Putem utiliza și instrucțiunea `break` cu etichetă. Astfel, putem scrie codul echivalent:

```
i = 1;
suma = 0;
eticheta: for ( ; ; ) {
    suma += i++;
```

```

    if (i > n) break eticheta;
}

```

Exemplul 2.9.8. Iată și un exemplu de utilizare a instrucțiunii continue în cadrul unei instrucțiuni for. Este vorba de calculul sumei numerelor întregi pozitive dintr-un tablou (cu 10 elemente).

```

public class SumaNumerelorPozitive {
    public static void main(String args[]) {
        int v[] = {3, 5, -3, 6, 2, -1, 8, 9, -4, 1}, i = 0,
            suma = 0;
        for (i = 0; i < 10; ++i) {
            if (v[i] < 0) continue;
            suma += v[i];
        }
        System.out.println("suma = " + suma);
    }
}

```

Ideea este simplă. Se parcurg toate elementele tabloului, iar pentru cele negative se sare peste instrucțiunea `suma += v[i];`. Astfel se va afișa la execuție 34.

2.9.8. Instrucțiunea break

Instrucțiunea de transfer de control necondiționat `break` are sintaxa generală:

```
break [<Identificator>;
```

unde `<Identificator>` este o etichetă opțională. O instrucțiune `break` fără etichetă transferă controlul celei mai interioare instrucțiuni `switch`, `while`, `do` sau `for` (care se mai numesc ținte `break` și care se termină normal). Astfel, o instrucțiune `break` fără etichetă întotdeauna se termină anormal, iar dacă nu există ținte `break` (adică vreo instrucțiune `switch`, `while`, `do` sau `for`) atunci se obține eroare de compilare.

O instrucțiune `break` cu eticheta `E` încearcă să transfere controlul instrucțiunii etichetate cu `E` (care se mai numește țintă `break` și care se termină normal) în care se află `break`. Ținta `break` nu trebuie neapărat să fie o instrucțiune `switch`, `while`, `do` sau `for`. Astfel, o instrucțiune `break` cu etichetă întotdeauna se termină anormal, iar dacă nu există ținte `break` cu eticheta `E`, atunci se obține eroare de compilare.

Așadar, chiar dacă instrucțiunea `break` are sau nu etichetă, aceasta se termină anormal.

Exemplele 2.6.3, 2.9.1, 2.9.3, 2.9.4, 2.9.7 conțin referiri la instrucțiunea `break`. Vom prezenta doar un exemplu simplu de folosire a lui `break` cu etichetă, dar în care ținta `break` nu este o instrucțiune `switch`, `while`, `do` sau `for`.

Exemplul 2.9.9. Fie programul Java de mai jos:

```

public class BreakIf {
    public static void main(String args[]) {

```

```

        boolean b = false;
        eticheta: if ((int)(Math.random() * 2) == 0) {
            b = true;
            break eticheta;
        }
        if (b) System.out.println("S-a executat break eticheta.");
        else
            System.out.println("Nu s-a executat break eticheta.");
    }
}

```

La unele execuții, programul de mai sus va afișa:

S-a executat break eticheta.

iar la altele:

Nu s-a executat break eticheta.

2.9.9. Instrucțiunea continue

Instrucțiunea continue poate apărea doar în interiorul unor instrucțiuni repetitive (`while`, `do`, `for`) și are sintaxa generală:

```
continue [<Identificator>;
```

unde `<Identificator>` se numește eticheta instrucțiunii continue. O instrucțiune continue fără etichetă încearcă transferarea controlului celui mai interior `while`, `do`, `for`, care se numesc ținte continue. Apoi termină imediat iterația curentă și începe una nouă. O instrucțiune continue se termină anormal, iar dacă nu există instrucțiunile `while`, `do` sau `for`, atunci se obține o eroare la compilare.

O instrucțiune continue cu eticheta `E` încearcă transferarea controlului instrucțiunii în care se află, dar care are eticheta `E`. Apoi termină imediat iterația curentă și începe una nouă. O instrucțiune continue se termină anormal, iar dacă nu există o instrucțiune `while`, `do` sau `for` etichetată cu `E` în care se află instrucțiunea continue, atunci se obține o eroare la compilare.

Așadar, întotdeauna o instrucțiune continue se termină anormal. Exemplul 2.9.8 conține referiri la instrucțiunea continue fără utilizarea etichetelor. Vom prezenta un exemplu simplu în care folosim continue cu etichete.

Exemplul 2.9.10. Această aplicație simplă pune în evidență suma elementelor de tip `double` ale unei matrice cu 3 linii și 4 coloane. La calculul sumei participă doar elementele care sunt mai mari în modul decât 0.01. Se va utiliza instrucțiunea continue care poate transfera necondiționat controlul programului la eticheta unui sau doi.

```

public class Continue {
    public static void main(String args[]) {

```

```
// m - numarul de linii, n - numarul de coloane
int m = 3, n = 4;
// i, j sunt variabile de lucru
int i, j;
// a este o matrice cu 3 linii si 4 coloane cu
// elemente de tip double
double [][] a = { {2.45, -5.67, 0.001, 5.32},
                  {5.65, 2.28, 4.01, 0.002},
                  {-0.005, 6.39, 8.13, -7.89}
                };

// suma este o variabila de tip double care tine
// suma elementelor mai mari in modul decat 0.01
double suma = 0;
unu: for (i = 0; i < m; i++) {
    doi: for (j = 0; j < n; j++) {
        if (Math.abs(a[i][j]) < 0.01)
            if (j < n - 1) continue doi;
        else continue unu;
        suma += a[i][j];
    }
}

System.out.println("Suma elementelor mai mari de 0.01"
    + " este " + suma);
}
```

Execuția sa va afișa textul:

```
Suma elementelor mai mari de 0.01 este 20.67
```

2.9.10. Instrucțiunea return

Instrucțiunea return cedează controlul apelului unei metode sau constructor și are sintaxa generală:

```
return [<Expresie>];
```

O instrucțiune return fără expresie trebuie să fie conținută în corpul unei metode care returnează void sau al unui constructor. O instrucțiune return nu poate apărea într-un inițializator static, deoarece acesta este executat la inițializarea clasei și nu se poate termina anormal. În schimb, instrucțiunea return încearcă transferul controlului către apelantul metodei sau constructorul acesteia. Mai precis, o instrucțiune return fără expresie se va termina anormal.

O instrucțiune return cu expresie trebuie să fie conținută într-o declarație de metodă care returnează o valoare, în caz contrar apare o eroare de compilare. <Expresie> trebuie să fie o variabilă sau valoare de tip T, iar tipul T trebuie să fie asignabil tipului

returnat de metodă, în caz contrar apare o eroare de compilare. La execuția instrucțiunii return <Expresie> se evaluează mai întâi <Expresie>. Dacă aceasta eșuează dintr-un anumit motiv, atunci instrucțiunea return se termină anormal din același motiv. Dacă evaluarea lui <Expresie> se termină normal și produce valoarea V, atunci instrucțiunea return se termină anormal, returnând valoarea V.

Așadar, instrucțiunea return se termină anormal întotdeauna.

Exemplele 2.4.8, 2.6.3, 2.8.8 și 2.9.5 se referă la instrucțiunea return. Vom prezenta un exemplu referitor la inițializatori statici.

Exemplul 2.9.11. Următorul program va furniza o eroare la compilare, deoarece avem o instrucțiune return într-un inițializator static:

```
public class InicializatorStatic {
    static int i = 6;
    static {
        System.out.println("Cod static: i = " + i++);
        return i;
    }
    public static void main(String[] args) {
        System.out.println("Cod in main(): i = " + i++);
    }
}
```

Evident, corectarea se poate realiza prin eliminarea instrucțiunii return i; din inițializatorul static. După corecție, programul va afișa la execuție:

```
Cod static: i = 6
Cod in main(): i = 7
```

2.9.11. Instrucțiunea throw

Termenul *excepție* din Java se referă la prevenirea apariției unei erori care are șanse mari de apariție (sau altfel spus se referă la tratarea cazurilor particulare). De exemplu, dacă dorim să încărcăm un fișier care nu există pe disc, dacă un fișier conține date corupte, dacă o conexiune de rețea eșuează, dacă există în program o încercare de accesare ilegală a memoriei, atunci trebuie să specificăm acest lucru. Excepțiile sunt indicate prin aruncarea (eng. *thrown*) lor și sunt detectate prin prinderea (eng. *catch*) acestora. Excepțiile sunt de fapt obiecte ale clasei `java.lang.Throwable`.

Instrucțiunea throw implică aruncarea unei excepții și are sintaxa generală:

```
throw <Expresie>;
```

Rezultatul este transferul imediat al controlului din care pot ieși instrucțiunile, constructorii, evaluările de inițializare a câmpurilor și apelurile metodelor până când se găsește o instrucțiune try care „prinde” (se unifică cu) valoarea aruncată. Dacă nu este găsită nici o instrucțiune try, atunci execuția firului responsabil cu throw este

terminată după apelul metodei `UncaughtException` pentru grupul de fire de execuție la care aparține firul în cauză. <Expresie> trebuie să fie o variabilă sau o valoare a unui tip referință asignabil la tipul `Throwable`, în caz contrar apare o eroare la compilare. În plus, una din următoarele trei condiții trebuie să aibă loc:

1. Dacă excepția nu este excepție verificată, atunci una din situații este adevărată:
 - a) Tipul lui <Expresie> este clasa `RuntimeException` sau o subclasă a sa;
 - b) Tipul lui <Expresie> este clasa `Error` sau o subclasă a sa.
2. Instrucțiunea `throw` este conținută într-un bloc `try` (a unei instrucțiuni `try` și tipul lui <Expresie> este asignabil cu tipul parametrului a cel puțin unei clauze `catch` a instrucțiunii `try`).
3. Instrucțiunea `throw` este conținută într-o declarație de metodă sau constructor și tipul lui <Expresie> este asignabil cel puțin unui tip listat în clauzele `throws`.

Instrucțiunea `throw` evaluează mai întâi <Expresie>. Dacă evaluarea lui <Expresie> se termină anormal dintr-un anumit motiv, atunci instrucțiunea `throw` se termină anormal din același motiv. Dacă evaluarea lui <Expresie> se termină normal, producând valoarea *V*, atunci instrucțiunea `throw` se termină anormal, aruncându-se valoarea *V*. Instrucțiunea `throw` se termină anormal mereu.

Exemplul 2.9.12. Reluăm metoda `parseInt()` din clasa `Integer`, care apare în Exemplele 2.5.18 și 2.6.3. Pentru a vedea ce excepție aruncă o metodă dintr-o bibliotecă Java ne uităm la prototipul ei. Codul metodei `parseInt()` arată cam așa (doar structura):

```
public static int parseInt(String s) throws
    NumberFormatException {
    ...
    /* codul metodei */
    ...
    throw new NumberFormatException();
    ... //etc
}
```

Exemplul 2.9.13. În acest exemplu, vom descrie o metodă care poate arunca excepții utilizând instrucțiunea `throw`.

```
import java.io.*;

public class TestExceptii {
    public static void main(String args[]) {
        for (int i = 0; i <= 2; i++) {
            try {
                System.out.println("\nTest caz " + i);
                metodal(i);
                System.out.println("Sfarsit caz " + i);
            }
        }
    }
}
```

```
catch (Exception e) {
    System.out.println("Exceptie:" + e.getMessage());
}
finally {
    System.out.println("Executam finally !");
}
}

private static void metodal(int i) throws Exception {
    System.out.println("Incepe metodal:");
    if (i != 0) throw new Exception("din metodal");
    System.out.println("Sfarsit metodal.");
}
} // sfarsitul definitiei clasei TestExceptii
```

La execuție, se va afișa:

```
Test caz 0
Incepe metodal:
Sfarsit metodal.
Sfarsit caz 0
Executam finally !
```

```
Test caz 1
Incepe metodal:
Exceptie:din metodal
Executam finally !
```

```
Test caz 2
Incepe metodal:
Exceptie:din metodal
Executam finally !
```

Se observă că în cazul când *i* are valoarea 0, nu se aruncă nici o excepție. În schimb, când *i* are valoarea 1 sau 2, `metodal()` nu se mai execută în totalitate, ci numai până la aruncarea excepției.

2.9.12. Instrucțiunea `synchronized`

Instrucțiunea `synchronized` va implica satisfacerea proprietății de excludere mutuală și anume executarea unui bloc de către cel mult un fir de execuție. După execuție, se eliberează resursa respectivă. Sintaxa generală a instrucțiunii `synchronized` este:

```
synchronized (<Expresie>) <Bloc>
```

Tipul lui <Expresie> trebuie să fie referință, în caz contrar apare o eroare la compilare. Instrucțiunea `synchronized` evaluează întâi <Expresie>. Dacă evaluarea se termină anormal dintr-un anumit motiv, atunci instrucțiunea `synchronized` se termină anormal din același motiv. Dacă valoarea lui <Expresie> este null, atunci se aruncă o excepție `NullPointerException`. Altfel, fie `V` valoarea lui <Expresie>. Firul de execuție închide resursa asociată lui `V`. Apoi se execută <Bloc>. Dacă execuția lui <Bloc> se termină normal, atunci resursa este eliberată și instrucțiunea `synchronized` se termină normal. Dacă execuția lui <Bloc> se termină anormal dintr-un anumit motiv, atunci resursa este eliberată, iar instrucțiunea `synchronized` se termină anormal din același motiv.

Închiderile resurselor realizate de instrucțiunea `synchronized` sunt la fel ca cele ale metodelor sincronizate (care utilizează modificatorul de acces `synchronized`). Un singur fir de execuție poate închide o resursă de mai multe ori.

Exemplul 2.9.14. Vom pune în evidență un exemplu simplu de închidere a unei resurse (un obiect al clasei respective).

```
public class InstructiuneaSynchronized {
    public static void main(String[] args) {
        InstructiuneaSynchronized is = new
            InstructiuneaSynchronized();
        synchronized(is) {
            synchronized(is) {
                System.out.println("Acum obiectul is este inchis de"
                    + "doua ori!");
            }
            System.out.println("Acum obiectul is este inchis o data!");
        }
        System.out.println("Acum obiectul is nu mai este inchis!");
    }
}
```

La execuție, acest program va afișa:

```
Acum obiectul is este inchis de doua ori!
Acum obiectul is este inchis o data!
Acum obiectul is nu mai este inchis!
```

2.9.13. Instrucțiunea try-catch-finally

Cum am prezentat în Secțiunea 2.9.11, excepțiile sunt indicate prin aruncarea lor, prin utilizarea instrucțiunii `throw` și sunt detectate prin prinderea lor, prin intermediul instrucțiunii `try-catch-finally`. Sintaxa generală a instrucțiunii `try-catch-finally` este:

```
try <BlocTry>
    [catch (<ParametruFormal1>) <BlocCatch1>]
```

```
[catch (<ParametruFormal2>) <BlocCatch2>]
...
[catch (<ParametruFormalN>) <BlocCatchN>]
[finally <BlocFinally>]
```

cu restricția că trebuie să existe o clauză `catch` sau `finally` (adică nu pot lipsi amândouă). O instrucțiune `try` poate avea mai multe clauze `catch` (numite și *rezolvitori de excepții*). Forma generală a lui <ParametruFormalK>, unde `K` este 1, 2, ..., `N`, este <TipK> <ParametruExcepțieK>, iar <TipK> trebuie să fie din clasa `Throwable` sau o subclasă a acesteia, în caz contrar apare o eroare la compilare. Domeniul variabilei parametru <ParametruExcepțieK> este <BlocCatchK>. Numele lui <ParametruExcepțieK> trebuie să difere de numele altor variabile locale sau parametri care au același domeniu, în caz contrar apare o eroare la compilare.

Dacă este aruncată o valoare și instrucțiunea `try` are mai multe clauze `catch` care o pot prinde, atunci controlul se transferă primei clauze `catch` cu care se unifică. Din acest motiv, în Java, blocul `catch` mai specific trebuie să le precedă pe cele mai generale, în caz contrar se obține eroare la compilare. Clauza `finally` se execută întotdeauna, chiar dacă se prinde vreo excepție sau nu.

Începem cu prezentarea lui `try-catch` (clauza `finally` va fi explicată ulterior). O instrucțiune `try` fără blocul `finally` începe prin execuția lui <BlocTry>. Atunci:

1. Dacă execuția lui <BlocTry> se termină normal, atunci nu se mai execută nimic în plus și spunem că instrucțiunea `try` s-a terminat normal;
2. Dacă execuția lui <BlocTry> nu se termină normal din cauza unei instrucțiuni `throw` de valoare `V`, atunci:
 - a) Dacă tipul lui `V` este asignabil lui <TipK>, unde `K` este 1, 2, ..., `N`, atunci se selectează prima clauză `catch` pentru care `V` se unifică cu <ParametruExcepțieK>, apoi se execută <BlocCatchK>. Dacă blocul <BlocCatchK> se termină normal, atunci instrucțiunea `try` se termină normal. Dacă blocul <BlocCatchK> se termină anormal dintr-un anumit motiv, atunci instrucțiunea `try` se termină anormal din același motiv.
 - b) Dacă tipul lui `V` nu este asignabil lui <TipK>, `K` este 1, 2, ..., `N`, atunci instrucțiunea `try` se termină anormal din cauza unei instrucțiuni `throw` de valoare `V`.
3. Dacă <BlocTry> se termină anormal dintr-un anumit motiv, atunci instrucțiunea `try` se termină anormal din același motiv.

Exemplul 2.9.15. Vom prezenta un program Java care compară aruncarea propriilor excepții față de excepțiile sistem (`RuntimeException`).

```
class ExceptiaNoastra extends Exception {
    ExceptiaNoastra() { }
    ExceptiaNoastra(String s) { super(s); }
}

public class Exceptie {
```



```

static void f() throws ExceptiaNoastra {
    throw new ExceptiaNoastra();
}

public static void main(String[] args) {
    try {
        f();
    }
    catch (RuntimeException re) {
        System.out.println("S-a aruncat exceptia: " + re);
    }
    catch (ExceptiaNoastra en) {
        System.out.println("S-a aruncat exceptia: " + en);
    }
} // sfarsitul definitiei clasei Exceptie

```

La execuție se va afișa:

S-a aruncat exceptia: ExceptiaNoastra

Exemplul 2.9.16. Următorul subprogram Java poate genera o împărțire (nepermisă) la 0 (va arunca o excepție `ArithmeticException`) sau o depășire de indice de vector (va arunca o excepție `ArrayIndexOutOfBoundsException`).

```

int x = (int) (Math.random() * 5);
int y = (int) (Math.random() * 10);
int [] z = new int[5];
try {
    System.out.println("y / x = " + (y / x));
    System.out.println("y = " + y + " z[y] = " + z[y]);
}
catch (ArithmeticException e) {
    System.out.println("Problema aritmetica: " + e);
}
catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Depasire de domeniu index vector " + e);
}

```

La execuție, dacă nu apare nici o excepție, atunci subprogramul își desfășoară fluxul normal. Dacă apare o excepție necaptată, atunci execuția subprogramului se oprește. Dacă există o instrucțiune `try`, dar nu există clauză `catch` corespunzătoare cu excepția apărută, se întrerupe execuția subprogramului. Dacă însă se găsește un `catch` corespunzător, atunci acesta se execută și se continuă execuția subprogramului cu instrucțiunea următoare construcției `try`.

Continuăm cu semantica instrucțiunii `try-catch-finally` (adică există blocul `finally`). O instrucțiune `try` cu bloc `finally` începe cu execuția lui `<BlocTry>`. Atunci:

1. Dacă execuția lui `<BlocTry>` se termină normal, atunci se execută `<BlocFinally>` și avem:
 - a) Dacă `<BlocFinally>` s-a terminat normal, atunci instrucțiunea `try` se termină normal;
 - b) Dacă `<BlocFinally>` s-a terminat anormal dintr-un motiv, atunci instrucțiunea `try` se termină anormal din același motiv.
2. Dacă execuția lui `<BlocTry>` nu se termină normal din cauza unei instrucțiuni `throw` de valoare `V`, atunci:
 - a) Dacă tipul lui `V` este asignabil lui `<TipK>`, iar `K` este 1, 2, ..., `N`, atunci se selectează prima clauză `catch` pentru care `V` se unifică cu `<Parametru ExceptieK>`, apoi se execută `<BlocCatchK>`.
 - i. Dacă blocul `<BlocCatchK>` se termină normal, atunci se execută `<BlocFinally>` și avem în continuare cazurile 1. a) sau 1. b).
 - ii. Dacă blocul `<BlocCatchK>` se termină anormal dintr-un anumit motiv `M1`, atunci se execută `<BlocFinally>` și dacă se termină normal, atunci instrucțiunea `try` se termină anormal din motivul `M2`, iar dacă `<BlocFinally>` se termină anormal din motivul `M2`, atunci instrucțiunea `try` se termină anormal din motivul `M2` (și nu din motivul `M1`).
 - b) Dacă tipul lui `V` nu este asignabil lui `<TipK>`, `K` este 1, 2, ..., `N`, atunci se execută `<BlocFinally>` și distingem situațiile:
 - i. Dacă `<BlocFinally>` se termină normal, atunci instrucțiunea `try` se termină anormal din cauza unei instrucțiuni `throw` de valoare `V`;
 - ii. Dacă `<BlocFinally>` se termină anormal din motivul `M1`, atunci instrucțiunea `try` se termină anormal din motivul `M1` (valoarea `V` este eliberată).
3. Dacă `<BlocTry>` se termină anormal din motivul `M1`, atunci se execută `<BlocFinally>` și distingem situațiile:
 - a) Dacă `<BlocFinally>` se termină normal, atunci instrucțiunea `try` se termină anormal din motivul `M1`;
 - b) Dacă `<BlocFinally>` se termină anormal din motivul `M2`, atunci instrucțiunea `try` se termină anormal din motivul `M2`.

Exemplul 2.9.17. Continuăm ideea din Exemplul 2.9.15 de a defini propria noastră clasă de excepții și punem în evidență utilizarea lui `finally`:

```

class ExceptiaNoastra extends Exception {
    ExceptiaNoastra() { }
    ExceptiaNoastra(String s) { super(s); }
}

public class ExceptieFinally {
    static void f() throws ExceptiaNoastra {

```

```

        throw new NullPointerException();
    }

    public static void main(String[] args) {
        try {
            f();
        }
        catch (ExceptiaNoastra re) {
            System.out.println("S-a aruncat exceptia: " + re);
        }
        finally {
            System.out.println("Exceptie neprinsa.");
        }
    }
} // sfarsitul definitiei clasei ExceptieFinally

```

La execuție, programul va afișa:

```

Exceptie neprinsa.
java.lang.NullPointerException
    at ExceptieFinally.f(ExceptieFinally.java:8)
    at ExceptieFinally.main(ExceptieFinally.java:13)

```

Clasa de excepții `NullPointerException` (derivată din `RuntimeException`) care este aruncată de metoda `f()` nu este prinsă de instrucțiunea `try` din `main()`, deoarece un obiect `NullPointerException` nu este asignabil unei variabile de tip `ExceptiaNoastra`. Aceasta implică execuția clauzei `finally` (care afișează prima linie), apoi firul de execuție asociat lui `main()` va afișa mesajele de eroare ulterioare.

În principiu, există patru situații când nu se execută instrucțiunile din corpul `finally`: oprirea firului de execuție curent, apelul `System.exit()`, oprirea calculatorului sau apariția unei excepții în blocul `finally`.

Exemplul 2.9.18. Să presupunem că un program deschide cu succes un fișier, dar apare o eroare în timpul citirii din fișier. Este de dorit închiderea fișierului în această situație. Pentru aceasta trebuie să utilizăm `finally`.

```

String linie;
try {
    while ((linie=fisier.readLine()) != null) {
        ...//procesare linie
    }
}
catch (IOException e) {
    campEroare.setText("Eroare in fisierul de intrare");
}

```

```

finally {
    fisier.close();
}

```

Astfel, fișierul curent va fi închis indiferent de excepția apărută.

2.10. Concluzii

Capitolul 2 se referă la fundamentele limbajului Java, în special la „aspectul static”, tipul referință urmând a fi prezentat în Capitolul 3.

Secțiunea 2 se referă la fișiere sursă. Acestea trebuie să posede extensia `.java` și să conțină cel mult o definiție a unei clase de bază publice, al cărei nume trebuie să coincidă cu numele fișierului (bineînțeles, fără extensie). Un fișier sursă poate conține un număr nelimitat de definiții de clase nepublice.

Secțiunea 3 se referă la atomi lexicali. Prima etapă a compilării unui program Java începe cu *analizarea lexicală* a codului sursă Java. Asta înseamnă parcurgerea secvențială a textului identificând cuvinte acceptate de sintaxa Java, numite *atomi lexicali* (eng. *tokens*). Acești atomi lexicali sunt: identificatori, cuvinte rezervate, literal, separatori și operatori ai limbajului Java. Spațiile și comentariile sunt de asemenea identificate, însă acestea nu mai sunt transmise copie programului Java în vederea continuării procesului de compilare (analiză sintactică, semantică etc). Înaintea prezentării traducerilor lexicale, se descriu caracterele în care sunt scrise programele Java, așa numitele *caractere Unicode*. Traducerea lexicală poate folosi orice secvență Unicode cu sens special (eng. *escape*) pentru a include orice caracter Unicode folosind doar caractere ASCII (eng. *American Standard Code for Information Interchange*).

Urmează secțiunea 4 referitoare la tipuri de date. Acestea trebuie cunoscute în momentul compilării pentru fiecare variabilă și expresie, fapt pentru care se spune că Java este un limbaj *puternic tipizat*. În Java, există trei categorii de tipuri de date: *primitive*, *referință* și *null*. Tipurile primitive sunt tipurile *numerice* și *boolean*. Tipurile numerice sunt tipurile *integrale* (`byte`, `short`, `int`, `long` și `char`) și tipurile în *virgulă flotantă* (`float` și `double`). Tipurile referință sunt tipurile *clasă*, *interfață* și *tablou*. Un obiect în Java este o instanță creată dinamic a unui tip clasă sau a unui tablou creat dinamic. Valorile unui tip referință sunt referințe la obiecte. Toate obiectele, inclusiv tablourile, suportă metode ale clasei `Object`. Literalii `String` sunt reprezentați de obiecte `String`. Tipul `null` este special, acesta neavând nume.

Secțiunea 5 se referă la expresii și operatori. Un aspect important în Java îl reprezintă *evaluarea expresiilor*, fie prin *efectele laterale* (asignarea de valori către variabile), fie pentru *valorile returnate* de acestea (utilizate ca argumente sau operanzi în alte expresii) sau pentru *afectarea secvenței de execuție* în instrucțiuni. Dacă o expresie este o variabilă sau o valoare, atunci expresia trebuie să aibă tipul cunoscut încă de la compilare. Ca și la variabile, valoarea unei expresii este compatibilă cu tipul expresiei. Astfel, valoarea unei expresii al cărei tip este `T` poate fi asignată unei variabile de tip `T`. Dacă tipul unei expresii este un tip primitiv, atunci valoarea

expresiei este de același tip primitiv. Dacă tipul expresiei este un tip referință, atunci clasa și valoarea obiectului referențiat nu se cunosc la momentul compilării, ci al execuției. Pentru a da definiția formală a *expresiei*, mai întâi definim noțiunea de *expresie primară*. Expresii primare sunt literalii, accesul la câmpuri și tablouri, apeluri de metode, expresii parantetizate. Un *operator* cu n argumente este o funcție cu n argumente care se mai numesc și *operandi*. Utilizând o manieră recursivă de descriere, o *expresie compusă* (sau simplu, *expresie*) poate fi expresie primară, o expresie care folosește operatori unari, binari și ternari sau o expresie de conversie. Prin cele 31 de exemple, secțiunea 5 prezintă expresiile și operatorii referitori la tipurile primitive de date, cele de tip referință urmând a fi prezentate în Capitolul 3.

Secțiunea 6 prezintă noțiunea de *variabilă* în Java. Aceasta este o locație de memorie care are un tip asociat (tip verificat la compilare). O variabilă conține o valoare care este compatibilă cu tipul variabilei. Valoarea unei variabile se poate schimba în urma unei asignări sau a operatorilor ++ și --. Compatibilitatea valorilor implicite și toate asignările unei variabile sunt verificate în timpul compilării, cu excepția tabloului, când verificarea se realizează în timpul execuției programului Java. Exemplul 2.6.3 pune în evidență o mare parte de tipuri de variabile, cele de tip instanță și de tip tablou urmând a fi prezentate în Capitolul 3.

Secțiunea 7 se referă la *declarații și inițializări*. Fiecare instrucțiune de declarare a unei variabile locale este conținută într-un bloc. Instrucțiunile de declarare a variabilelor locale pot fi intercalate printre alte instrucțiuni ale blocului. O declarare de variabilă locală poate apărea și într-o instrucțiune *for*, caz în care se execută la fel ca o instrucțiune de declarare a unei variabile locale. O instrucțiune de declarare a unei variabile locale este o instrucțiune executabilă. La execuție, declaratorii sunt procesați de la stânga la dreapta. Dacă un declarator are o expresie de inițializare, atunci expresia este evaluată și valoarea se asignează variabilei. Dacă declaratorul nu are o expresie de inițializare, atunci compilatorul Java verifică dacă fiecare referință către variabilă este precedată de execuția unei asignări a variabilei. Dacă nu este așa, atunci se semnalează o eroare la compilare. Fiecare inițializare (cu excepția primei) este executată dacă precedentă expresie de inițializare se evaluează normal. Execuția unei declarații a unei variabile locale se termină normal dacă evaluarea ultimei expresii de inițializare se face normal. Evident, dacă declarația unei variabile locale nu conține expresii de inițializare, atunci execuția acesteia se termină întotdeauna normal.

Secțiunea 8 prezintă noțiunea de *conversie*. Aceasta are rolul de a modifica tipul unei expresii într-un anumit context. În fiecare context, sunt permise doar câteva conversii specifice: conversii *identice*, conversii *primitive implicite* („de la mic la mare”, eng. *widening conversions*), conversii *primitive explicite* („de la mare la mic”, eng. *narrowing conversions*), conversii de *referință implicite*, conversii de *referință explicite*, conversii la *String*. Cele 11 exemple prezentate în secțiunea 8 se referă la conversiile identice, cele primitive (implicite și explicite) și conversii la *String*. În Capitolul 4 vor fi prezentate conversiile de *referință*.

Secțiunea 9 prezintă noțiunea de *structuri de control (instrucțiuni)*. Acestea au rolul de a controla execuția unui program Java prin efectul lor și nu pentru faptul că întorc valori (cum era cazul expresiilor, Secțiunea 2.5). Anumite instrucțiuni conțin alte

instrucțiuni (sau expresii) ca parte a structurii lor. Instrucțiunile Java sunt: instrucțiunea *vidă*, etichetele, expresiile, *if*, *switch*, *while*, *do*, *for*, *break*, *continue*, *return*, *throw*, *synchronized*, *try*. Spre deosebire de alte limbaje de programare, în Java nu există instrucțiunea *goto*. Cele 18 exemple ale acestei secțiuni pun în evidență aceste instrucțiuni.

2.11. Test grilă

Întrebarea 2.11.1. Fie următoarea secvență de cod Java:

```
1) float a = 2;
2) float b = 1.3;
```

Care afirmație este adevărată?

- a) Instrucțiunile 1 și 2 sunt corecte.
- b) Instrucțiunea 1 este corectă, instrucțiunea 2 este incorectă.
- c) Instrucțiunile 1 și 2 sunt incorecte.

Întrebarea 2.11.2. Fie următoarea declarație Java:

```
public private int h;
```

Care afirmație este adevărată?

- a) Variabila *h* va fi accesată în mod *public*, deoarece se ia în considerare primul modificador de acces.
- b) Variabila *h* va fi accesată în mod *private*, deoarece se ia în considerare ultimul modificador de acces.
- c) Va fi eroare la compilare, deoarece o variabilă nu poate fi în același timp accesată *public* și *private*.

Întrebarea 2.11.3. Fie următorul subprogram Java:

```
int x = 0;
if (Double.isInfinite(2 / x))
    System.out.println("infinit");
else
    System.out.println("2 / 0");
```

Ce puteți spune despre acesta?

- a) Eroare la compilare din cauza împărțirii la zero.
- b) Eroare la execuție din cauza împărțirii la zero (se aruncă o excepție *Arithmetic Exception*).
- c) Programul este corect și va afișa: *infinit*.
- d) Programul este corect și va afișa: *NaN*.

Întrebarea 2.11.4. La ce valoare se va evalua următoarea expresie:

```
2f<3d?+0.0:5<7?0.3:21
```

- a) Expresia conține erori sintactice.
- b) Unele conversii trebuie realizate explicit.
- c) Se va evalua la 0.0.
- d) Se va evalua la 0.3.

Întrebarea 2.11.5. Fie următorul subprogram Java:

```
String s1 = "anul" + 200 + 2, s2 = 200 + 2 + "anul";
System.out.println("s1 = " + s1 + ", s2 = " + s2);
```

Ce va afișa la execuția sa?

- a) s1 = anul202, s2 = 202anul
- b) s1 = anul2002, s2 = 202anul
- c) s1 = anul202, s2 = 2002anul
- d) s1 = anul2002, s2 = 2002anul
- e) Eroare la execuție: este necesară o conversie explicită!

Întrebarea 2.11.6. Fie următorul subprogram Java:

```
int as = 3, bs = 2, cs = 4;
System.out.print(((as < bs++) & (cs++ < bs)) + " ");
System.out.println(as + " " + bs + " " + cs);
System.out.print(((as < bs++) && (cs++ < bs++)) + " ");
System.out.println(as + " " + bs + " " + cs);
```

Ce se poate spune despre acesta?

- a) Eroare la compilare: nu se poate aduna o valoare booleană cu un String;
- b) Subprogramul se compilează și la execuție afișează:

false 3 3 5

false 3 4 4

- c) Subprogramul se compilează și la execuție afișează:

false 3 3 5

false 3 4 5

- d) Subprogramul se compilează și la execuție afișează:

false 3 3 5

false 3 5 6

Întrebarea 2.11.7. Fie următorul program Java:

```
public class Asignare {
    public static void main(String args[]) {
        int a = 3;
        int b = (a = 2) * a;
        int c = b * (b = 5);
        System.out.println("a = " + a + ", b = " + b + ", c = " + c);
    }
}
```

Ce va afișa acesta la execuție?

- a) a = 2, b = 4, c = 20
- b) a = 2, b = 5, c = 20 ✓
- c) a = 2, b = 5, c = 25
- d) a = 3, b = 6, c = 30

Întrebarea 2.11.8. Fie următorul subprogram Java:

```
System.out.println((1<2)?5:(3<4)) + " ";
```

Ce se poate spune despre acesta?

- a) Eroare la compilare: nu se poate converti o valoare booleană la un int.
- b) Subprogramul se compilează și la execuție afișează 5.
- c) Subprogramul se compilează și la execuție afișează 37.
- d) Subprogramul se compilează și la execuție afișează true.

Întrebarea 2.11.9. Fie următorul subprogram Java:

```
double d = 2.95;
int i = 4;
System.out.println(++d>i?d:i);
```

Ce se poate spune despre acesta?

- a) Eroare la compilare: nu putem converti o valoare double la int.
- b) Subprogramul se compilează și la execuție afișează 3.95.
- c) Subprogramul se compilează și la execuție afișează 4.
- d) Subprogramul se compilează și la execuție afișează 4.0.

Întrebarea 2.11.10. Fie următorul apel Java:

```
System.out.println(1 < 2 < 3);
```

Ce se poate spune despre acesta?

- a) Eroare la compilare.
- b) Eroare la execuție.
- c) Apelul se compilează și la execuție afișează true.
- d) Apelul se compilează și la execuție afișează false.

Întrebarea 11. Fie următorul program Java:

```
public class Apel {
    static int f(short a) { return a * 2; }
    public static void main(String[] args) {
        System.out.println(f(3));
    }
}
```

Ce se poate spune despre acesta?

- a) Eroare la compilare: nu există o definiție a metodei f() cu parametrul int.

- b) Eroare la compilare: metoda `f()` nu întoarce o valoare de tip `short`.
 c) Programul se compilează și la execuție afișează 6.

Întrebarea 2.11.12. Fie următorul subprogram Java:

```
byte b = -7 >>> 1;
System.out.println(b);
```

Ce se poate spune despre acesta?

- a) Eroare la compilare.
 b) Eroare la execuție.
 c) Subprogramul se compilează și la execuție afișează -3.
 d) Subprogramul se compilează și la execuție afișează -4.

Întrebarea 2.11.13. Fie următorul subprogram Java:

```
byte b = 7 >>> 1;
System.out.println(b);
```

Ce se poate spune despre acesta?

- a) Eroare la compilare.
 b) Eroare la execuție.
 c) Subprogramul se compilează și la execuție afișează 3.
 d) Subprogramul se compilează și la execuție afișează -3.

Întrebarea 2.11.14. Fie următorul program Java:

```
public class Program {
    static int x = 6;
    public static void main(String[] args) {
        System.out.print("x = " + x);
        int x = (x = 3) * x;
        System.out.print(", x = " + x);
    }
}
```

Ce se poate spune despre acesta?

- a) Eroare la compilare: variabila `x` este declarată de două ori.
 b) Programul se compilează și la execuție afișează `x = 6, x = 3`.
 c) Programul se compilează și la execuție afișează `x = 6, x = 9`.
 d) Programul se compilează și la execuție afișează `x = 6, x = 18`.

Întrebarea 2.11.15. Fie următorul program Java:

```
public class Program {
    public static void main(String[] args) {
        float min = Float.NEGATIVE_INFINITY;
        float max = Float.POSITIVE_INFINITY;
```

```
        System.out.println((int)(char)min + " " + (int)(char)max);
    }
}
```

Ce se poate spune despre acesta?

- a) Eroare la compilare: conversii imposibile.
 b) Programul se compilează și la execuție afișează 0 255.
 c) Programul se compilează și la execuție afișează 0 65535.
 d) Programul se compilează și la execuție afișează 0 -1.

Întrebarea 2.11.16. Fie următorul subprogram Java:

```
int a = 3;
if (a++ < 4)
    if (++a < 4)
        System.out.println(a);
    else
        System.out.println(a);
```

Ce se poate spune despre acesta?

- a) Eroare la compilare.
 b) Subprogramul se compilează și la execuție afișează 4.
 c) Subprogramul se compilează și la execuție afișează 5.
 d) Subprogramul se compilează și la execuție nu afișează nimic.

Întrebarea 2.11.17. Fie următorul program Java:

```
public class Program {
    static void f(int k) {
        switch (k) {
            default: System.out.print("i "); break;
            case 1: System.out.print("1 "); break;
            case 2: case 3: System.out.print("23 "); break;
            case 4: case 5: System.out.print("45 ");
        }
    }
    public static void main(String[] args) {
        for (int i = 0; i < 6; i++)
            f(i);
    }
}
```

Ce se poate spune despre acesta?

- a) Eroare la compilare.
 b) Programul se compilează și la execuție afișează `i 1 23 23 45 45`.
 c) Programul se compilează și la execuție afișează `i 1 23 45`.
 d) Programul se compilează și la execuție afișează `i 1 23 23 45 45 i`.

Întrebarea 2.11.18. Fie următorul subprogram Java:

```
int i = 1, suma = 0;
for ( ; ; ) {
    suma += i++;
    if (i > 5) break;
}
System.out.print(suma);
```

Ce se poate spune despre acesta?

- Eroare la compilare: lipsesc părțile componente ale lui for.
- Subprogramul se compilează și la execuție afișează 0.
- Subprogramul se compilează și la execuție afișează 10.
- Subprogramul se compilează și la execuție afișează 15.

Întrebarea 2.11.19. Fie următorul program Java:

```
public class Program {
    public static void main(String args[]) {
        int v[] = {2, 4, -2, 8, -2}, i = 0, suma = 0;
        for (i = 0; i < 10; ++i) {
            if (v[i] < 0) continue;
            suma += v[i];
        }
        System.out.println("suma = " + suma);
    }
}
```

Ce se poate spune despre acesta?

- Eroare la compilare.
- Programul se compilează și la execuție afișează 0.
- Programul se compilează și la execuție afișează 14.
- Programul se compilează și la execuție afișează -4.

Întrebarea 2.11.20. Fie următorul program Java:

```
public class Subiecte {
    public static void main(String args[]) {
        float f = 4.50;
        System.out.println(f);
    }
}
```

Ce se poate spune despre acesta?

- Eroare la compilare: nu se poate converti implicit 4.50 la float.
- Programul se compilează și la execuție afișează 4.5.
- Programul se compilează și la execuție afișează 4.50.
- Programul se compilează și la execuție afișează 4.5000000.

2.12. Exerciții propuse spre implementare

Exercițiul 2.12.1. Scrieți un program Java care să convertească mile terestre în km (și invers). Formula de transformare este 1milă = 1.6 km.

Exercițiul 2.12.2. Scrieți un program Java care să împacheteze o dată calendaristică într-o variabilă de tip int. Furnizați și o funcție de despachetare. De exemplu, 20 februarie 2002 se reprezintă ca un întreg astfel: primii 5 biți rezervați pentru zi, următorii 4 biți pentru lună și ultimii pentru an.

Exercițiul 2.12.3. Scrieți o clasă Java care generează numere pseudoaleatoare din domeniul 0...65535. Presupunând că avem o valoare inițială (sămânță, eng. *seed*) r_0 , pentru generarea următorului număr aleator din domeniul 0...65535 utilizăm formula $r_{\text{Nou}} = ((r_{\text{vechi}} * 25173) + 13849) \% 65536$.

Exercițiul 2.12.4. Scrieți un program Java care primește la intrare un număr de secunde și întoarce numărul maxim de ore, de minute, de secunde care este echivalent ca timp cu numărul inițial de secunde. De exemplu, 7384 secunde este echivalent cu 2 ore, 3 minute și 4 secunde.

Exercițiul 2.12.5. Folosind o buclă while și fără a folosi tablouri, scrieți un program Java care calculează termenul al n-lea din *șirul lui Fibonacci*. Șirul lui Fibonacci este dat de recurența liniară de ordin 2:

$$\begin{aligned} a_1 &= a_2 = 1; \\ a_{n+2} &= a_{n+1} + a_n, \quad n = 1 \end{aligned}$$

Generalizare la recurențe liniare de ordin k , $k \neq 1$, cu coeficienți arbitrari.

Exercițiul 2.12.6. Scrieți un program Java care să calculeze cel mai mare divizor comun dintre a și b (notat cu (a,b)), unde a, b sunt numere întregi, folosind algoritmul lui *Euclid*. De exemplu, $(12, 8) = 4$, deoarece se pot aplica împărțiri succesive astfel: $12 = 8 * 1 + 4$, $8 = 4 * 2 + 0$.

Exercițiul 2.12.7. Să se scrie un program Java care să calculeze $n!$ ($n! = 1 * 2 * \dots * n$), unde $0 < n < 13$ este un număr natural. Determinați $n!$ și pentru $n \geq 13$.

Exercițiul 2.12.8. Folosind structura for, scrieți un program Java care calculează următoarele formule logice (sub forma unei tabele de adevăr): $b_1 \mid b_3 \mid b_5$ și $b_1 \& b_2 \mid b_4 \& b_5$, unde b_1, b_2, b_3, b_4, b_5 sunt variabile logice (cu valori false, true).

Exercițiul 2.12.9. Să se scrie o metodă Java care să returneze minimul a trei numere de același tip primitiv (folosind operatorul condițional $?$: sau instrucțiunea if). Folosind această metodă, scrieți o metodă Java care să returneze minimul a patru numere.

Exercițiul 2.12.10. Fie funcția lui *Collatz*: $f(n) = n/2$ dacă n este par, și $f(n) = 3 * n + 1$ dacă n este impar. Să se scrie un program Java care determină k natural minim astfel încât $f(f(\dots f(n)\dots)) = 1$, unde parantezele $()$ apar de k ori. De exemplu, pentru $n = 6$ avem: $f(6) = 3$, $f(3) = 10$, $f(10) = 5$, $f(5) = 16$, $f(16) = 8$, $f(8) = 4$, $f(4) = 2$, $f(2) = 1$, deci k natural minim este 8.

Exercițiul 2.12.11. Scrieți un program Java care calculează suma divizorilor naturali ai unui număr natural n . Un număr este *perfect*, dacă este egal cu suma divizorilor proprii pozitivi (de exemplu: $28 = 1 + 2 + 4 + 7 + 14$). Să se genereze primele 4 numere perfecte.

Exercițiul 2.12.12. Presupunem că depunem o sumă (depozit la termen) într-o bancă care oferă o dobândă de 25 % (de exemplu) pe an. Să se calculeze suma finală după un anumit număr de ani (se va face capitalizarea contului, adică se va ține cont de „dobânda la dobândă”).

Exercițiul 2.12.13. Folosind operatori pe biți, scrieți metode Java care:

- testează dacă un număr de tip `short/int/long` este divizibil cu 2, 4, 8. Generalizare (divizibilitate cu 2^n);
- testează dacă un număr este pozitiv sau negativ;
- calculează pentru un n dat, multiplii de 2, 4, ... Generalizare;
- calculează pentru un n dat, $[n/2]$, $[n/4]$, ... Generalizare.

2.13. Proiecte propuse spre implementare

Proiectul 2.13.1. (Jocul *cap-pajură*, folosind o simulare Monte-Carlo) Presupunem că dispunem de o monedă ideală (nemăsluită). Doi jucători aruncă cu moneda după următoarele reguli:

- Se fac un număr total de n aruncări.
- Primul jucător aruncă moneda și celălalt spune *cap* sau *pajură*.
- Dacă acesta *ghicește* care parte a monedei va pica, atunci se inversează jucătorii (adică aruncă al doilea și primul încearcă să ghicească).
- La sfârșit, trebuie afișat scorul (și procentul de câștig al fiecăruia).

Proiectul 2.13.2. (Conjectura lui *Goldbach*) Orice număr par mai mare decât 2 se poate scrie ca sumă a două numere prime. Scrieți un program Java care verifică această conjectură pentru numere situate între m și n . De exemplu, dacă $m=700$ și $n=1100$, atunci trebuie afișat:

```
700 = 17 + 683
702 = 11 + 691
704 = 3 + 701
...
1098 = 5 + 1093
1100 = 3 + 1097
```

Generalizare: Scrieți toate combinațiile posibile de adunare a două numere prime egale cu un număr dat.

Proiectul 2.13.3. (Jocul „*hârtie, pumn, foarfece*”) Presupunem că avem doi jucători care folosesc mâna dreaptă pentru reprezentarea a trei obiecte: *hârtie* înseamnă palma întinsă, *pumn* înseamnă mâna strânsă sub formă de pumn și *foarfece* înseamnă două degete depărtate (semnul victoriei). Ei își arată simultan mâna dreaptă în una din aceste configurații (de mai multe ori). Dacă ei arată același lucru, este remiză (nu câștigă nimeni). Dacă nu, se aplică una dintre următoarele trei reguli:

- Hârtia acoperă pumnul (deci palma întinsă câștigă față de pumn).
- Pumnul sparge foarfecele.
- Foarfecele taie hârtia.

Să se simuleze acest joc, făcând un număr arbitrar de evenimente și precizând scorul final. Se cere să se joace persoană-calculator și, varianta a doua, calculator-calculator.

Proiectul 2.13.4. O *ruletă* este o mașină care selectează la întâmplare un număr între 0 și 35. Jucătorul poate paria pe un număr par/impar sau poate paria pe un număr oarecare. Câștigul unui pariu par/impar se premiază cu 2/1 euro respectiv, cu excepția cazurilor în care ruleta alege 0. Dacă jucătorul ghicește numărul selectat de ruletă, atunci este plătit cu 35/1 euro. Să se simuleze acest joc, știind că jucăm pariuri de 1 euro și având o sumă inițială de n euro.

3. Clase, interfețe și tablouri

3.1. Cuvinte cheie

- clase, atribute, metode, constructori, modificatori de acces
- interfețe, metode abstracte, clase abstracte
- tablouri
- conversii ale tipului referință

3.2. Clase

Clasa este entitatea de bază a unui limbaj orientat spre obiecte. În Java este obligatorie definirea a cel puțin unei clase. Nu este însă obligatorie crearea unei instanțe (unui obiect) a acelei clase, caz în care se vor utiliza date și metode statice.

O declarație de clasă în limbajul Java are sintaxa generală:

```
[<ModificatoriClasa>] class <Identificator> [extends <NumeClasa>]
[implements <ListaInterfete>] {
    [<DeclaratiiCorpClasa>]
}
```

unde:

- <ModificatoriClasa> pot fi: public, abstract, final (cel mult o singură apariție și, de preferat, în această ordine);
- <Identificator> reprezintă numele clasei;
- <NumeClasa> este numele unei alte clase (denumită clasă de bază sau superclasă);
- <ListaInterfete> este o listă de nume de interfețe de forma: <Nume Interfata1>, <NumeInterfata2>, ..., <NumeInterfataN>;
- <DeclaratiiCorpClasa> este o listă de declarații de forma:
 - <DeclaratiiMembruClasa> care la rândul ei poate fi <Declarație Atribut> sau <DeclarațieMetoda>;
 - <InitializatorStatic>;
 - <DeclaratiiConstructor>.

Dacă o clasă este declarată într-un pachet numit Pachet, atunci numele complet (calificat) al clasei este Pachet.<Identificator> (se poate consulta Exemplul 2.2.1). În caz contrar, dacă definim clasa într-un pachet fără nume, atunci numele complet al clasei este <Identificator> (se pot consulta Exemplele 2.4.8, 2.4.9 etc.).

Un constructor al unei clase este o metodă care are numele respectivei clase. Constructorii nu returnează nimic și nu au tip, nici măcar void. Constructorii sunt apelați automat în momentul creării unui obiect din clasa respectivă și sunt utilizați pentru inițializarea datelor membre.

Dacă pentru o clasă nu declarăm nici un constructor, atunci la compilare se creează automat un constructor implicit care este public, nu are nici un parametru și execuția sa nu are nici un efect. A nu se confunda constructorul implicit (cel generat automat) cu cel fără nici un parametru (scris de programator). Dacă declarăm un constructor, atunci nu se mai generează constructorul implicit.

Destructorul este metoda care se apelează automat la distrugerea unui obiect. Destructorii sunt extrem de utilizați în limbajul de programare C++, mai ales pentru eliberarea resurselor ocupate de respectivul obiect. În Java lucrurile stau altfel datorită colectorului de gunoarie, care are drept sarcină eliberarea automată a memoriei la care aplicația nu mai are nici o referință. Prin suprascriserea metodei finalize() din clasa Object, programatorul poate specifica o mulțime de acțiuni, care vor fi executate o dată cu distrugerea obiectului.

Fie clasa Cerc:

```
class Cerc {
    // date member
    int raza, x, y;
    // constructor fara parametri
    Cerc() {
        raza = 1;
        x = 20;
        y = 20;
    }
    // constructor cu parametri
    Cerc(int r, int a, int b) {
        raza = r;
        x = a;
        y = b;
    }
    // metoda publica
    public void setRaza(int r) { raza = r }
} // terminarea clasei Cerc
```

Aceasta conține trei date membre de tip int: raza, x și y, doi constructori, unul cu parametri și unul fără parametri și o metodă publică setRaza().

3.2.1. Domeniul de vizibilitate al numelui unei clase

Domeniul de vizibilitate al numelui unei clase este întreg pachetul din care aceasta face parte.

Exemplul 3.2.1. Programul de mai jos intitulat DV.java face parte din pachetul cu numele Pachet. Constructorul clasei DV va crea două obiecte din clasele C1, respectiv C2. Spre deosebire de exemplele precedente, clasa C1 conține un obiect din clasa C2 și invers. Deoarece domeniul de vizibilitate a unei clase este pachetul în care se află, rezultă că putem referi un obiect din clasa C1 în clasa C2 (și reciproc). Dacă nu ar fi fost conținute în același pachet, am fi obținut eroare la compilare.

```
package Pachet;

public class DV {
    public DV(int k) { // constructor
        C2 cDoi = new C2(k);
        System.out.println(cDoi);
        C1 cUnu = new C1(cDoi);
        System.out.println(cUnu);
    }
}
```

```

class C1 {
    /* date membre */
    int x;
    C2 c2;
    /* constructor */
    C1(C2 c2) {
        this.x = c2.x;
        this.c2 = c2;
    }
    /* definirea unei metode */
    public String toString() {
        return "x = " + x + " c2 = " + c2;
    }
}

```

```

class C2 {
    /* date membre*/
    int x;
    C1 c1;
    /* constructor */
    C2(int x) {
        this.x = ++ x;
        this.c1 = new C1(this);
    }
    /* definirea unei metode */
    public String toString() {
        return "x = " + x;
    }
}

```

De exemplu, putem scrie un program de tip aplicație de sine stătătoare care va apela clasa DV (declarată public, deci accesibilă):

```

import Pachet.*;

public class DomeniuVizibilitate {
    public static void main(String args[]) {
        DV obiect = new DV(4);
    }
}

```

În urma execuției programului de mai sus se obține:

```

x = 5
x = 5 c2 = x = 5

```

3.2.2. Modificatorii unei clase

În definiția clasei, modificatorii unei clase, care sunt notați cu <ModificatoriClasa>, pot fi: public, abstract, final (de preferat, în această ordine). Aceștia pot avea cel mult o singură apariție, în caz contrar apare o eroare la compilare.

Dacă o clasă este declarată public, atunci aceasta poate fi accesată de orice cod Java care poate accesa pachetul în care este definită această clasă. Dacă o clasă nu este declarată public, atunci aceasta poate fi accesată doar din pachetul în care este declarată.

O clasă este *abstractă* dacă aceasta este incomplet definită (conține o metodă neimplementată). Doar clasele abstracte pot avea metode abstracte (neimplementate). Dacă o clasă conține o metodă abstractă și nu este declarată abstract, atunci se obține o eroare la compilare. O clasă C are metode abstracte în unul din cazurile:

1. Clasa C conține o declarație a unei metode abstracte;
2. Clasa C moștenește o metodă abstractă de la o altă clasă;
3. Când există o interfață care implementează clasa C și care declară sau moștenește o metodă (care este clar că este abstractă, deoarece o interfață nu are metode definite) și clasa C nu declară și nici nu moștenește o metodă care o implementează.

Exemplul 3.2.2. În continuare prezentăm un program Java care definește o clasă derivată dintr-o clasă abstractă și o interfață.

```

// definirea unei clase abstracte
abstract class C1 {
    // declararea datelor membre
    int x = 1, y = 2;
    // declararea unei metode
    void f(int x) {
        this.x += x;
        g(x);
    }
    // declararea unei metode abstracte
    abstract void g(int y);
}

// definirea unei interfete
interface I1 {
    public void h();
}

/* definirea unei metode care deriveaza clasa C1
si implementează interfata I1 */

```

```

class C2 extends C1 implements I1 {
    int z = 10;
    // implementarea metodei abstracte din clasa C1
    void g(int y) {
        this.y += y;
    }
    // implementarea metodei mostenita din interfata I1
    public void h() {
        System.out.println("x = " + x + ", y = " + y + ", z = " + z);
    }
}

// definirea clasei care va utiliza un obiect de tip C2
public class Testare {
    public static void main(String args[]) {
        C2 obiect = new C2();
        obiect.f(4);
        obiect.g(5);
        obiect.h();
    }
}

```

Precizăm faptul că o interfață grupează un set de metode abstracte. Aceste metode sunt implicit abstracte, deci nu trebuie să mai fie precedate de cuvântul cheie `abstract`.

Clasa `C1` trebuie declarată `abstract`, deoarece conține metoda `g()` nedefinită. Aceasta este definită în clasa derivată `C2`. Interfața `I1` conține metoda abstractă `h()` al cărei cod va fi definit în clasa `C2` care implementează `I1` (în caz contrar, ar fi trebuit să definim clasa `C2` abstractă). În clasa de aplicație `Testare`, dacă am fi încercat crearea unui obiect din clasa `C1` sau din interfața `I1`, am fi obținut eroare la compilare.

În principiu, un tip clasă se declară `abstract` doar dacă se intenționează definirea altor subclase care să definească restul metodelor nedefinite. Dacă însă intenția este să prevenim instanțierea clasei, putem proceda altfel, și anume declarând un singur constructor fără argumente declarat `private` (și nu `public`).

De obicei, clasele abstracte sunt utilizate pentru a reprezenta concepte abstracte, de exemplu o componentă grafică. Apoi se derivează respectiva clasă pentru a reprezenta elemente concrete, de exemplu un buton.

Exemplul 3.2.3. Programul Java de mai jos:

```

class C {
    int x = 10;
    private C() { x = 5; }
}

```

```

public class TestareInstantiere {
    public static void main(String args[]) {
        C obiect = new C(); // eroare la compilare
    }
}

```

implică eroare la compilare, deoarece se încearcă instanțierea unei clase care are un constructor `private`.

O clasă care are doar un constructor și acesta este declarat `private` nu poate fi nici derivată. În acest caz, clasa derivată nu poate apela constructorul clasei de bază, deoarece acesta este `private`.

O altă modalitate de a preveni derivarea claselor este utilizând modificatorul `final`. O clasă poate fi declarată `final`, dacă definiția sa este completă și nu sunt necesare definiri de subclase. Încercarea de definire a unei clase derivate dintr-o clasă declarată `final` conduce la o eroare la compilare. În plus, o clasă definită `final` și `abstract` conduce de asemenea la o eroare de compilare. Deoarece o clasă declarată `final` nu poate avea subclase, metodele acesteia nu pot fi niciodată definite.

Exemplul 3.2.4. Iată cum se definește o clasă `final` standard Java:

```

public final class C extends Object

```

Astfel, clasa `C` nu va putea fi moștenită, dar poate fi instanțiată, cum se poate vedea în programul:

```

final class C {
    int x = 5;
}

public class ClasaFinal {
    public static void main(String args[]) {
        C obiect = new C();
        System.out.println(obiect.x);
    }
}

```

care va afișa la execuție 5.

3.2.3. Clase derivate

Clauza opțională `extends` din declarația unei clase (vezi sintaxa generală în secțiunea 3.2) specifică care este clasa imediat următoare clasei curente (este vorba de `<NumClasa>`, care se mai numește *superclasă*). Mai precis, în declarația `C1 extends C2`, spunem că `C2` este *superclasa directă* (se mai numește *clasa de bază*) a lui `C1`, iar `C1` este *subclasa directă* (se mai numește *clasa derivată*) a lui `C2`. Singura clasă care nu are *superclasă*

este `Object` (din pachetul `java.lang`). Clasa `Object` este superclasa tuturor claselor Java. Dacă o declarație de clasă nu folosește explicit clauza `extends`, atunci clasa `Object` este superclasa directă. Clasa `C2` trebuie să fie accesibilă, altfel se obține eroare la compilare (`C1` este în același pachet cu `C2` sau `C2` este declarată `public` într-un pachet accesibil). La fel, dacă `C2` este declarată `final`, atunci se obține o eroare la compilare (clasele declarate `final` nu pot avea subclase).

Se observă în Java că o clasă se poate deriva din cel mult o altă clasă, spre deosebire de alte limbaje de programare orientate obiect (cum ar fi `C++`) care permit *derivarea multiplă* (adică o clasă se poate deriva din mai multe clase). Spre deosebire de limbajul `C++`, unde relația de derivare se poate asocia cu un graf, în Java relația de derivare se poate asocia unui arbore cu rădăcina etichetată cu `Object`.

Relația de *derivare a claselor* (indirectă) este închiderea tranzitivă a relației de derivare directă. Spunem că C_1 este *derivată* din C_N dacă există clasele $C_2, C_3, \dots, C_{N-1}$ astfel încât C_1 este derivată direct din C_2 , C_2 este derivată direct din C_3 , ..., C_{N-1} este derivată direct din C_N . Clasele $C_1, C_2, \dots, C_N$ sunt distincte, în caz contrar se obține eroare la compilare. Evident, derivarea directă este un caz particular de derivare indirectă (când $N=1$).

Exemplul 3.2.5. În programul Java de mai jos:

```
class C1 {
    int x;
}

class C2 extends C1 {
    int y;
}
```

avem următoarele relații de derivare directă:

- clasa `C1` este clasă derivată direct din `Object` și clasa `Object` este superclasa directă a lui `C1`;
- clasa `C2` este clasă derivată direct din `C1` și clasa `C1` este superclasa directă a lui `C2`;
- din a) și b), rezultă că `C2` este derivată (indirect) din clasa `Object`.

Exemplul 3.2.6. Dacă în definiția unei clase derivate apare clasa însăși, atunci se formează un circuit și se obține eroare la compilare, cum este cazul programului Java de mai jos, când clasa `C1` este derivată direct din clasa `C2` și clasa `C2` este derivată direct din clasa `C1`. Astfel, `C1` este derivată (indirect) din `C1`, lucru care conduce la eroare la compilare.

```
class C1 extends C2 {
    int x;
}

class C2 extends C1 {
    int y;
}
```

3.2.4. Clasa `Object`

În Java clasa `Object` este superclasă pentru toate clasele. Orice clasă din Java moștenește metodele din clasa `Object`. Cele mai importante sunt:

Prototipul metodei	Descrierea metodei
<code>protected Object clone()</code>	Creează și returnează o copie a obiectului curent.
<code>public boolean equals(Object obj)</code>	Testează dacă obiectul specificat este echivalent (egal) cu cel curent.
<code>protected void finalize()</code>	Această metodă este apelată de către <i>garbage collector</i> când se constată de către acesta că nu mai există referințe la acest obiect.
<code>public Class getClass()</code>	Returnează clasa din care face parte obiectul curent în momentul execuției.
<code>public int hashCode()</code>	Returnează codul asociat obiectului curent.
<code>public String toString()</code>	Returnează reprezentarea obiectului curent într-un <code>String</code> .

Metodele `notify()`, `notifyAll()` și `wait()` vor fi discutate în capitolul destinat firelor de execuție. Clasa `Object` posedă doar un constructor fără parametri.

Exemplul 3.2.7. Utilizarea metodei `getClass()`:

```
public class Object1 {
    public static void main(String args[]) {
        // crearea unui obiect de tip String
        String s = "Sir";
        // crearea unei referințe Object
        Object o = s;

        // afisarea numelui clasei din momentul execuției
        System.out.println(o.getClass().getName());
    }
}
```

Execuția programului de mai sus va afișa la consolă:

```
java.lang.String
```

Chiar dacă apelul este realizat de o referință de tip `Object`, este afișat numele clasei efective a obiectului.

Fiecărei instanțe a unei clase i se asociază un cod unic pentru o execuție a programului. Codul asociat se obține cu metoda `hashCode()`. Metoda `equals()` testează dacă obiectul curent și cel specificat au același cod. Dacă dorim ca pentru obiectele unei anumite clase să testeze altceva decât codul obiectului, atunci vom redefini metoda.

Exemplul 3.2.8. Testarea egalității codurilor aferente obiectelor:

```

public class Object2 {
    public static void main(String[] args) {
        // crearea a doua obiecte
        String s = "sir 1";
        Object o = new String("sir 2");

        // cele doua sunt diferite
        if (o.equals(s))
            System.out.println("Egalitate");
        else
            System.out.println("Diferenta");

        // acum devin identice
        o = s;
        // ambele sunt referinta la acelasi obiect
        if (o.equals(s))
            System.out.println("Egalitate");
        else
            System.out.println("Diferenta");
    }
}

```

Programul de mai sus va afișa:

```

Diferenta
Egalitate

```

Metoda `clone()` poate arunca excepția `CloneNotSupportedException` în cazul în care clasa nu implementează interfața `Cloneable`. Aceasta nu conține nici o metodă. De regulă, metoda `clone()` creează un nou obiect cu aceleași valori ale datelor membre, iar modificarea originalului sau a copiei nu va afecta și celălalt obiect. Deoarece metoda `clone()` este protejată, aceasta nu poate fi apelată din exteriorul clasei. De obicei este apelată din interiorul claselor derivate.

Exemplul 3.2.9. Clonarea claselor se poate realiza în maniera următoare:

```

// definirea unei clase simple care permite clonarea
class ObiectSimplu implements Cloneable {
    // declararea unei date membre
    private String content = "";

    public void setContent(String s) {
        content=s;
    }
}

```

```

public String getContent() {
    return content;
}

// supradefinirea metodei clone()
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        System.err.println("Eroare la clonare!");
        return this;
    }
}

public class Object3 {
    public static void main(String[] args) {
        // crearea unui obiect
        ObiectSimplu ob1 = new ObiectSimplu();
        // clonarea unui obiect
        ObiectSimplu ob2 = (ObiectSimplu) ob1.clone();

        // verificarea celor doua obiecte
        System.out.println(ob1.equals(ob2));
        System.out.println(ob1.getContent()==ob2.getContent());
        // modificarea originalului
        ob1.setContent("Test");
        // verificarea continutului celor doua obiecte
        System.out.println(ob1.getContent()==ob2.getContent());
    }
}

```

Programul anterior creează o copie a unui obiect, testează dacă într-adevăr s-a realizat o copie, după care se verifică dacă cele două obiecte sunt independente. Rezultatul va fi:

```

false
true
false

```

Prima linie denotă faptul că avem două obiecte, iar a doua că ambele posedă același conținut. Ultima linie afișată arată că modificarea unui obiect nu are efect asupra celui alt.

Metoda `toString()` este apelată automat ori de câte ori se cere o conversie la tipul `String`. Frecvent se utilizează în operația de concatenare a șirurilor de caractere.

Exemplul 3.2.10. Adăugăm la clasa `ObiectSimplu` metoda `toString()`:

```
public String toString() {
    return "ObiectSimplu: " + content;
}
```

Următorul program:

```
public class Object4 {
    public static void main(String[] args) {
        // crearea unui obiect
        ObiectSimplu ob = new ObiectSimplu();
        // setarea continutului
        ob.setContent("Test");
        // afisarea mesajului
        // metoda toString() va fi apelata implicit
        System.out.println(ob + "\t\t[ Ok ]");
    }
}
```

va afișa la execuție:

```
ObiectSimplu: Test      [ Ok ]
```

3.2.5. Implementarea interfețelor

O interfață este un tip special de dată care conține un grup de date membre și de metode abstracte. Fiecare clasă care implementează o interfață va trebui să redefinească respectivele metode.

Clauza opțională `implements` din declarația unei clase (vezi sintaxa generală în secțiunea 3.2) specifică lista interfețelor imediat următoare clasei curente (este vorba de <ListaInterfete>, care se mai numesc *superinterfețe*). Mai precis, în declarația `C1 implements I1, I2, ..., IN`, spunem că `I1, I2, ..., IN` sunt *superinterfețe directe*. Acestea trebuie să fie accesibile, altfel se obține eroare la compilare (`C1` este în același pachet cu `I1, I2, ..., IN` sau `I1, I2, ..., IN` sunt declarate public într-un pachet accesibil). Numele interfețelor `I1, I2, ..., IN` trebuie să fie distincte, altfel se obține eroare la compilare.

Dacă o clasă implementează o interfață, atunci respectiva clasă trebuie să redefinească toate metodele interfeței (în caz contrar se obține eroare la compilare). Există unele clase speciale numite *adaptor* care implementează o interfață definind toate metodele acesteia. De cele mai multe ori acestea nu au nici un efect. Programatorul, dacă dorește să realizeze o clasă care să implementeze o interfață care conține mai multe metode, va extinde adaptorul corespunzător interfeței și va defini doar metodele care prezintă interes.

Ca și în cazul claselor, relația de *derivare a interfețelor* (indirectă) este închiderea tranzitivă a relației de derivare directă a interfețelor (vom reveni în secțiunea 3.3).

Spunem că I_1 este *derivată* din I_N dacă există clasele $I_2, I_3, \dots, I_{N-1}$ astfel încât I_1 este derivată direct din I_2 , I_2 este derivată direct din I_3 , ..., I_{N-1} este derivată direct din I_N . Interfețele $I_1, I_2, \dots, I_N$ sunt distincte, în caz contrar se obține eroare la compilare.

Folosind relația de derivare a interfețelor, spunem că un tip interfață I este *superinterfață* a tipului clasă C dacă I este *superinterfață directă* a lui C sau există I' *superinterfață directă* a lui C astfel încât I' este derivată din I .

Exemplul 3.2.11. În programul Java de mai jos:

```
// declararea interfeței I1
interface I1 {
    // declararea unei date membre
    int x = 0;
    // declararea unei metode (care este abstractă)
    public int f();
}

// declararea interfeței I2
// aceasta va mosteni datele si metodele membre din I1
interface I2 extends I1 {
    int y = 1;
    public int g();
}

// declararea clasei C1
class C1 {
    int z = 2;
    public int h() { return z; }
}

/* declararea clasei C2 care extinde clasa C1
   si implementeaza interfata I2 */
class C2 extends C1 implements I2 {
    int t = 3;
    // redefinirea metodelor din interfata I2
    public int f() { return x; }
    public int g() { return y; }
}

public class SuperInterfete {
    public static void main(String args[]) {
        // instantierea unui obiect din clasa C2
        C2 obiect = new C2();
        System.out.println("x = " + obiect.f() +
```

```

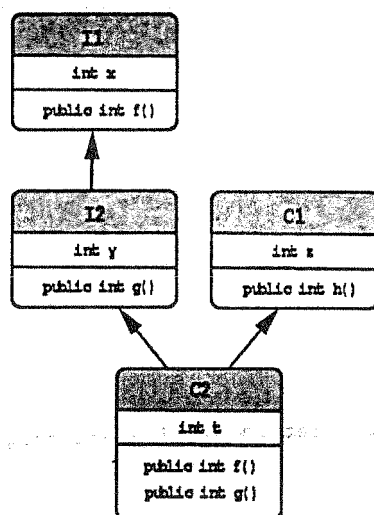
        ", y = " + obiect.g() + ", z = " + obiect.h()
        + ", t = " + obiect.t);
    }
}

```

avem următoarele relații de derivare între clase și interfețe:

- interfața I2 este derivată din I1, adică I1 este superinterfață pentru I2;
- clasa C2 este derivată din clasa C1 și din interfața I2, adică C1 este superclasă a lui C2 și I2 este superinterfață a lui C2;
- conform cu a) și b), rezultă că C2 este derivată (indirect) din I1, adică I1 este superinterfață (indirectă) pentru C2.

Această ierarhie de clase și interfețe este ilustrată și în figura de mai jos:



Este obligatoriu pentru C2 (care nu este abstractă) să implementeze metodele f() și g(), în caz contrar se obține eroare la compilare. Programul de mai sus va afișa la execuție:

```
x = 0, y = 1, z = 2, t = 3
```

3.2.6. Declarațiile membrilor unei clase

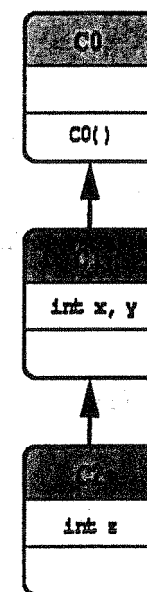
Corpul unei clase poate conține declarații ale membrilor clasei, adică *atribute* și *metode*. Corpul clasei poate conține și alte declarații de ne-membri, cum sunt inițializatorii statici și constructorii. Domeniul numelui unui membru declarat într-o clasă sau moștenit de o clasă este corpul declarat în acea clasă. Membrii moșteniți ai unui tip clasă sunt membri moșteniți din superclasa directă (cu excepția lui Object, care nu are superclasă) și membri moșteniți din toate superinterfețele directe. Membrii private

ai unei clase nu sunt moșteniți de alte subclase ale acelei clase. Doar membrii `protected` sau `public` sunt moșteniți de subclase declarate în alt pachet. Deoarece inițializatorii statici și constructorii nu sunt membri, aceștia nu se moștesc.

Dacă o clasă nu are nici un constructor, atunci se apelează implicit un constructor fără argumente pentru clasa curentă, care conține un apel al constructorului implicit din superclasa acestuia. Acest lucru se realizează explicit prin intermediul cuvântului rezervat `super`, care este valid doar în metode, instanțe, constructori sau inițializatori ai unei variabile instanță a unei clase. Doar în clasa `Object` nu se poate utiliza `super`, deoarece aceasta nu are superclase. Astfel, dacă avem `class C2 extends C1` și clasa `C2` nu are constructori, atunci se creează un constructor implicit echivalent cu `C2() { super(); }`. Dacă apelul lui `super` într-un constructor este explicit, atunci acesta trebuie să fie prima instrucțiune în constructor. În caz contrar, se obține eroare la compilare.

Exemplul 3.2.12. Următorul program Java pune în evidență accesul clasei derivate la clasa de bază folosind apelul implicit al lui `super()`. Dacă superclasa directă nu are nici ea constructor explicit, atunci se va apela constructorul implicit al acesteia ș.a.m.d., până la găsirea unei superclase care are constructor implicit sau fără argumente. În caz contrar, se obține eroare la compilare.

Ierarhia de clase din acest exemplu este:



```

class C0 {
    C0() {
        System.out.println("Am apelat constructorul C0()");
    }
}

```

```

class C1 extends C0 {
    int x = 1, y;
}

class C2 extends C1 {
    int z = 3;
}

public class TestDoi {
    public static void main(String[] args) {
        C2 obiect = new C2();
        System.out.println(obiect.x + obiect.y + obiect.z);
    }
}

```

La execuție se va afișa:

```

Am apelat constructorul C0()
4

```

Clasa C2 nu are constructor, deci se va crea constructorul implicit cu codul C2() {super();}, apelându-se constructorul lui C1. Deoarece nici această clasă nu are constructor, se va crea constructorul implicit cu codul C1() {super();}, apelându-se constructorul lui C0. Acum se va afișa mesajul Am apelat constructorul C0(). Valorilor atributelor obiect.x, obiect.y și obiect.z sunt 1, 0, 3 (atributele neinițializate explicit de programator se inițializează cu 0). Astfel, se va afișa și suma acestora.

Exemplul 3.2.13. Următorul program Java pune în evidență accesul clasei derivate la clasa de bază prin apelul explicit al lui super().

```

// definirea clasei C1
class C1 {
    int x = 1;
    // definirea constructorilor
    C1() {
        f();
    }
    C1(int x) {
        this.x = x;
    }
    // definirea unei metodei
    void f() {
        System.out.print("x = " + x);
    }
}

```

```

// definirea clasei C2 care extinde C1
class C2 extends C1 {
    int y;
    // definirea constructorului
    C2(int x, int y) {
        // apelarea explicita a constructorului superclasei
        super(x);
        this.y = y;
    }
    // definirea unei metode membre
    void g() {
        System.out.println("y = " + y);
        f();
    }
}

```

```

public class Test {
    public static void main(String[] args) {
        // crearea unui obiect de tip C2
        C2 obiect = new C2(2, 3);
        // invocarea metodei membre g()
        obiect.g();
    }
}

```

La execuție se va afișa:

```

y = 3
x = 2

```

La crearea unui obiect din clasa C2 se va apela constructorul explicit cu două argumente, care, la rândul lui, prin intermediul lui super(), apelează constructorul clasei C1 cu un singur parametru. Urmează afișarea celor două valori atribut.

3.2.6.1. Niveluri de acces

Există patru niveluri de acces: public, protected, implicit și private.

Dacă tipul clasă sau interfață este declarat public, atunci acesta poate fi accesat de orice cod Java care accesează pachetul în care este declarat. În caz contrar, acesta poate fi accesat doar din pachetul în care acesta este declarat.

O dată sau o metodă membră a unui tip referință (clasă, interfață sau tablou) sau un constructor al unui tip clasă este accesibil doar dacă tipul este accesibil și membrul sau constructorul permite accesul astfel:

- Dacă membrul sau constructorul este declarat public, atunci se permite accesul. Toți membrii unei interfețe sunt implicit public;

- b) Altfel, dacă membrul sau constructorul este declarat `protected`, atunci accesul este permis doar dacă:
 - i. Accesul la membru sau constructor apare din pachetul clasei în care este declarat membrul `protected`;
 - ii. Accesul apare dintr-o subclasă a clasei în care este declarat membrul `protected` (detalii mai jos);
- c) Altfel, dacă membrul sau constructorul este declarat `private`, atunci accesul este permis doar dacă acesta apare din clasa în care este declarat;
- d) Altfel, este vorba de *acces implicit*, și este permis doar dacă accesul apare din pachetul în care este declarat tipul.

Continuăm cu detalii despre `protected`. Un membru sau constructor `protected` al unui obiect poate fi accesat din afara pachetului în care acesta este declarat doar de codul responsabil de implementarea acelui obiect. Astfel, fie C1 clasa în care este declarat membrul sau constructorul `protected` și fie C2 subclasa în care apelăm/accesăm acest membru sau constructor `protected`. Atunci:

1. Dacă este vorba de un membru `protected` (atribut sau metodă), pe care îl notăm cu M, avem:
 - a) Dacă accesul se face printr-o expresie de forma `super.M`, atunci acesta este permis;
 - b) Dacă accesul se face printr-un nume calificat `C.M`, unde C este o clasă, atunci accesul este permis doar dacă C este C2 sau o subclasă a lui C2;
 - c) Dacă accesul se face printr-un nume calificat `C.M`, unde C este un nume de expresie, atunci accesul este permis doar dacă tipul expresiei C este C2 sau o subclasă a lui C2;
 - d) Dacă accesul se face printr-o expresie de acces de câmp `E.M`, unde E este o expresie primară sau un apel de metodă `E.M()`, atunci accesul este permis doar dacă E este C2 sau o subclasă a lui C2.
2. Dacă este vorba de un constructor `protected`, avem:
 - a) Dacă accesul se face printr-un apel de constructor a unei superclase (folosind `super()`), atunci accesul este permis;
 - b) Dacă accesul se face printr-o expresie de creare a unei instanțe (folosind `new C()`), atunci accesul nu este permis;
 - c) Dacă accesul se face printr-un apel al metodei `newInstance()` a clasei `Class`, atunci accesul nu este permis.

Exemplul 3.2.14. Exemplul de mai jos pune în evidență faptul că în exteriorul pachetului P1, clasa C2 nu este accesibilă, nefiind declarată `public`. În schimb, atributul `x` și metoda `f()` din clasa C1 vor putea fi accesate din orice pachet, deoarece sunt declarate `public`.

```
package P1;

public class C1 {
    public int x = 1;
```

```
    public void f(int x) {
        this.x += x;
    }
}

class C2 {
    C1 obiectC1;
}
```

Astfel, următorul program va accesa, din afara pachetului P1, atributul și metoda clasei C1. Dacă am fi încercat crearea unui obiect din clasa C2, am fi obținut eroare la compilare, deoarece clasa C2 nu este declarată `public`, deci nu există acces în afara pachetului P1.

```
import P1.*;

public class Test {
    public static void main(String[] args) {
        C1 obiect = new C1();
        obiect.f(2);
        System.out.println(obiect.x);
    }
}
```

Programul de test este corect și va afișa la execuție: 3.

Continuăm cu un exemplu în care punem în evidență accesul la atribute `protected`.

Exemplul 3.2.15. Fie următorul program care face parte din pachetul p1:

```
package p1;

public class C1 {
    protected int x = 2;
    public void f(int x) {
        this.x += x;
    }
    public int getX() {
        return x;
    }
}
```

Clasa C1 este declarată `public`, deci este accesibilă și din afara pachetului p1. Metodele `f()` și `getX()` sunt declarate `public`, deci sunt accesibile tuturor obiectelor de tip C1. Atributul `x` este declarat `protected`, deci este accesibil doar subclaselor clasei C1, acesta putând fi accesat (citit/modificat) doar prin intermediul funcțiilor de acces (`f()` și `getX()`). Astfel, programul:

```
import p1.*;
```

```

public class Test {
    public static void main(String[] args) {
        C1 obiect = new C1();
        obiect.f(2);
        System.out.println(obiect.getX());
    }
}

```

va afișa la execuție 4.

Un exemplu în care punem în evidență accesul implicit la atribute este următorul:

Exemplul 3.2.16. Fie următoarele programe care fac parte din pachetul p1 (în subdirectorul C:\temp\p1\):

```

package p1;

public class C1 {
    protected int x = 2;
    public void f(int x) {
        this.x += x;
    }
    public void g() {
        System.out.println("x = " + x);
    }
}

```

și respectiv clasa C2, derivată a lui C1, din pachetul p1 (în subdirectorul C:\temp\p1\):

```

package p1;

public class C2 extends C1 {
    int y = 2;
    public void f(int x, int y) {
        this.x += x;
        this.y += y;
    }
    public void g() {
        super.g();
        System.out.println("y = " + y);
    }
}

```

În directorul curent (în subdirectorul C:\temp\), considerăm clasa C3 derivată a lui C2. Accesul la metoda f() din clasa C2 se face folosind cuvântul rezervat super. Dacă se încearcă accesarea directă (cum am accesat atributul x în clasa C2) a metodei f(), atunci s-ar obține eroare la compilare.

```

import p1.C2;

class C3 extends C2 {
    int z = 3;
    public void f(int x, int y, int z) {
        super.f(x, y);
        this.z += z;
    }
    public void g() {
        super.g();
        System.out.println("z = " + z);
    }
}

```

Considerăm acum și programul de aplicație (în subdirectorul C:\temp\):

```

import p1.*;

public class Test {
    public static void main(String[] args) {
        C3 obiect = new C3();
        obiect.f(2, 3, 4);
        obiect.g();
    }
}

```

care va afișa la execuție:

```

x = 3
y = 5
z = 7

```

Exemplul 3.2.17. Evidențierea accesului la atribute private:

```

class C1 {
    private int x = 1;
    public void f(int x) {
        this.x += x;
    }
    private static int n;
    void afisareN() {
        System.out.println(n);
    }
}

class C2 extends C1 {
    private int y = 2;
}

```

```

    void f(int x, int y) {
        super.f(x);
        this.y += y;
        //      n++;
    }
}

public class Test {
    public static void main(String[] args) {
        C2 object = new C2();
        object.f(2, 3);
        object.afisareN();
    }
}

```

Accesarea lui `n` în clasa `C2` conduce la eroare la compilare, deoarece atributul `n` al clasei `C1` este `private`, deci accesibil doar în clasa `C1`. Execuția programului de mai sus va afișa 1.

3.2.6.2. Declarațiile atributelor unei clase

<DeclarațieAtribut> (din definiția declarării unei clase, secțiunea 3.2) are sintaxa generală:

```
[<ModificatorAtribut>] <Tip> <DeclarațieVariabilăI> [, . . . ,
<DeclarațieVariabilăN>];
```

unde:

1. <ModificatorAtribut> poate fi: `public`, `protected`, `private`, `final`, `static`, `transient`, `volatile`;
2. <DeclarațieVariabilăI>, cu $I = 1, \dots, N$, poate fi:
 - a) <Identificator> [= <InițializatorVariabilă>] sau
 - b) <Identificator> []...[] [= <InițializatorTablou>]

În cazul declarării unui atribut de tip tablou, nu trebuie confundate parantezele pătrate (`[]`) cu convenția de notație a unei părți opționale. <Identificator> va fi numele atributului declarat. Numele unui atribut are domeniul în întreg corpul clasei în care este declarat acel atribut. Într-o declarație a unei clase, atributele trebuie să aibă nume distincte, în caz contrar se obține eroare la compilare. Atributele și metodele pot avea același nume, din moment ce ele se utilizează în contexte diferite. Dacă o clasă declară un atribut cu un anumit nume, se spune că acel atribut *ascunde* toate declarațiile atributului cu același nume din superclasele și superinterfețele acelei clase. O clasă moștenește din superclasa directă și din superinterfețele directe toate atributele superclasei și superinterfețelor care sunt accesibile codului clasei și nu sunt ascunse în declarația clasei. Este posibil ca o clasă să moștenească mai multe atribute

cu același nume. Un atribut ascuns poate fi accesat prin nume calificat (dacă acesta este `static`), printr-o expresie de acces a atributului folosind cuvântul rezervat `super` sau prin conversie explicită la tipul superclasei.

Exemplul 3.2.18. Programul Java de mai jos pune în evidență câteva dintre modalitățile de acces al atributelor ascunse ale unei clase. Se observă că atributul `x` și metoda `x()` sunt declarate atât în clasa `C1`, cât și în clasa `C2`. Acestea sunt accesate în clasa `TestAtribute` fără a se obține eroare la compilare. Atributul static `y` poate fi accesat prin adăugarea numelui clasei la care ne referim.

```

class C1 {
    private int x = 1;
    public static int y = 3;
    void x() {
        System.out.println(x);
    }
}

class C2 extends C1 {
    private int x = 2;
    public static int y = 4;
    void x() {
        super.x();
        System.out.println(x);
    }
}

public class TestAtribute {
    public static void main(String[] args) {
        C2 object = new C2();
        object.x();
        System.out.println("C1.y = " + C1.y);
        System.out.println("C2.y = " + C2.y);
    }
}

```

Execuția acestui program va afișa:

```

1
2
C1.y = 3
C2.y = 4

```

Continuăm cu prezentarea modificatorilor atributelor. În subsecțiunea 3.2.5.1, au fost prezentați modificatorii `public`, `private`, `protected`, precum și cel implicit.

Dacă un atribut este declarat `static`, atunci se alocă o singură dată memorie pentru acesta, indiferent de numărul de instanțe ale clasei respective. Deoarece toate instanțele unei clase au același atribut (cu aceeași valoare în orice moment al execuției programului), atributele statice se mai numesc *variabile clasă*. În contrast, un atribut ne-static se numește *variabilă instanță*. Când se creează o nouă instanță a clasei, se creează și o nouă variabilă asociată atributului ne-static. Atributele statice sunt inițializate în momentul încărcării clasei, deci înaintea creării instanțelor acelei clase.

Exemplul 3.2.19. Fie declarația clasei `Increment`:

```
class Increment
{
    static int x = 0;
    Increment() { x++; } // constructorul
}
```

Variabila `x` este statică. Asta înseamnă că oricâte instanțe ale clasei `Exemplu` sunt create, există doar un `x` (pentru care s-a alocat memorie doar o singură dată). De altfel, și inițializarea cu 0 are loc în momentul încărcării clasei. Variabila statică este incrementată de fiecare dată când se apelează constructorul clasei, astfel fiind posibil să știm câte obiecte ale clasei `Increment` s-au creat.

Există două moduri de referire la o variabilă statică:

- printr-o referință la orice obiect al clasei;
- prin numele clasei (aceasta este varianta recomandată).

Exemplul 3.2.20. Continuăm Exemplul 3.2.19 punând în evidență „eventualele confuzii” care pot apărea dacă utilizăm prima formă:

```
Increment e1 = new Increment();
Increment e2 = new Increment();
System.out.println("e1.x = " + e1.x);
e1.x = 100;
e2.x = 200;
System.out.println("e1.x = " + e1.x);
```

Prima dată se afișează `e1.x = 2`, iar a doua oară `e1.x = 200`. Justificarea constă în faptul că obiectele `e1` și `e2` (instanțe ale clasei `Exemplu`) au aceeași zonă de memorie pentru atributul static `x`. Cu alte cuvinte, `e1.x` și `e2.x` se referă la aceeași variabilă (statică).

Atributele statice au multe aplicații, cum ar fi numărarea instanțelor unei clase.

Exemplul 3.2.21. Reluăm Exemplul 3.2.19 pentru a pune în evidență o modalitate de numărare a instanțelor unei clase. Astfel, declarăm un atribut static `numarInstante`, care va fi incrementat în constructor. La fiecare instanțiere a clasei `Increment`,

se va apela constructorul `Increment()` și deci se va incrementa variabila clasă `numarInstante`.

```
public class NumarareInstante {
    public static void main(String args[]) {
        System.out.println("S-au creat " + Increment.numarInstante +
            " instante ale clasei Increment");
        Increment a = new Increment();
        System.out.println("S-au creat " + Increment.numarInstante +
            " instante ale clasei Increment");
        Increment b = new Increment();
        System.out.println("S-au creat " + Increment.numarInstante +
            " instante ale clasei Increment");
    }
}

class Increment {
    static int numarInstante = 0;
    Increment() {
        numarInstante++;
    }
}
```

La execuție, acest program va afișa:

```
S-au creat 0 instante ale clasei Increment
S-au creat 1 instante ale clasei Increment
S-au creat 2 instante ale clasei Increment
```

Prezentăm în continuare atributul `final`. Declarațiile atributelor `final` (atât ale variabilelor clasă, cât și ale celor instanță) trebuie să conțină o expresie de inițializare, în caz contrar apare o eroare de compilare. Valoarea unui atribut `final` este aceeași pe toată execuția programului. Orice asignare ulterioară a unui atribut `final` conduce la o eroare de compilare. Dacă un atribut `final` va fi egal cu o referință la un obiect, atunci se poate schimba starea obiectului, dar atributul se va referi mereu la același obiect. Declararea unui atribut `final` poate servi la documentarea programului, la evitarea erorilor de programare și poate face mai ușoară pentru compilator generarea codului.

Exemplul 3.2.22. Următorul program pune în evidență utilizarea unui atribut `final`.

```
class C1 {
    int x;
    C1(int x) {
        this.x = x;
    }
}
```

```

void setX(int x) {
    this.x = x;
}

final static C1 y = new C1(1);
}

public class AtributFinal {
    public static void main(String args[]) {
        System.out.println(C1.y);
        C1 obiect = new C1(4);
        System.out.println(obiect.y);
        obiect.setX(6);
        System.out.println(obiect.y);
    }
}

```

Atributul `y` trebuie declarat și `static` pentru că altfel, la crearea primului obiect din clasa `C1`, se va crea o instanță a clasei și pentru atributul `y` al clasei `C1`, care la rândul ei va genera iar o instanță asociată atributului `y` ș.a.m.d.. Astfel, execuția programului va intra într-un fel de „recursie” și va umple memoria de instanțe ale clasei `C1`. În schimb, declarând atributul `y` ca fiind `static`, vom avea o singură valoare pentru variabila clasă `y` pentru toate instanțele clasei `C1`. Programul va afișa la execuție un rezultat de genul (la execuții la momente de timp diferite se pot obține alte adrese de memorie):

```

C1@3179c3
C1@3179c3
C1@3179c3

```

Astfel, chiar dacă se schimbă starea obiectului (modificând valoarea atributului `x`), valoarea atributului `y` va fi mereu aceeași.

Variabilele descrise `final` sunt de fapt constante.

Exemplul 3.2.23. Un exemplu de definire a unui atribut constant este cel utilizat pentru transformarea unității de măsură inci în centimetri.

```

final float CmInch = 2.54f;

```

Reamintim faptul că litera `f` de la sfârșitul numărului indică faptul că acesta este considerat de tip `float`.

Variabilele `final static` sunt folosite pentru declararea constantelor care sunt utilizate într-un număr de clase diferite într-un program.

Exemplul 3.2.24. Putem considera o clasă care conține doar constante, astfel:

```

public class Constante {

```

```

    public final static float C=3.0E+8f;
    public final static int LATIME=640;
    ...
}

```

Prin convenție, ca și în C/C++, numele constantelor se scriu cu majuscule.

Exemplul 3.2.25. Constanta `PI` este declarată în clasa `Math` astfel:

```

public final static double PI;

```

Continuăm cu prezentarea variabilelor `transient`. Variabilele marcate `transient` nu fac parte din starea persistentă a unui obiect. Acestea se utilizează la serializarea obiectelor. *Serializarea* este procesul de împărțire a datelor în octeți astfel încât să poată fi trimiși printr-un flux de ieșire. La implementarea unei clase trebuie ținut cont care parte a stării clasei trebuie serializată. Implicit, toate datele nestatice și neocasionale (non-`transient`) vor fi serializate. Aceasta înseamnă că dacă nu se dorește serializarea unor anumite variabile, atunci acestea trebuie declarate `transient`. Vom reveni asupra modificadorului `transient` în Capitolul 5.

Exemplul 3.2.26. În cadrul clasei `C1` de mai jos:

```

class C1 {
    int x;
    transient int y;
}

```

doar atributul `x` va putea fi salvat și serializat, deoarece nu este declarat `transient`.

Prezentăm mai jos variabilele `volatile`. Limbajul Java permite firelor de execuție care accesează variabile să păstreze copii de lucru ale acestora; aceasta permite o implementare mai eficientă a firelor de execuție multiple. Aceste copii de lucru trebuie să fie corelate cu originalele din memoria principală numai în punctele de sincronizare prescrise, adică la blocarea (eng. *lock*) și deblocarea obiectelor (eng. *unlock*). Ca o regulă, pentru a fi siguri că aceste variabile partajate sunt consistente și actualizate, un fir de execuție trebuie să se asigure că are folosire exclusivă a acestor variabile prin blocarea lor.

Modificadorul `volatile` permite specificarea unei variabile să fie modificată în orice moment de către firele de execuție. Scopul modificadorului `volatile` este protecția împotriva coruperii memoriei unei variabile. Într-un mediu asincron, coruperea datelor apare când variabila este memorată în registrii procesorului. Modificadorul `volatile` determină sistemul de execuție Java să refere o variabilă direct din memorie, în loc de a utiliza registre. În plus, variabila este citită din memorie și scrisă înapoi în memorie după fiecare acces la memorie. Modificadorul `volatile` se utilizează foarte rar, preferându-se în schimb utilizarea lui `synchronized`.

Java pune la dispoziție posibilitatea declarării atributelor `volatile`, caz în care copiile de lucru trebuie să fie corelate cu originalele din memoria principală de fiecare

dată când se accesează variabila. Mai mult, operațiile originale pentru variabilele volatile sunt procesate de memoria principală în exact aceeași ordine ca și cea cerută de firul de execuție. Vom reveni cu detalii în capitolul dedicat firelor de execuție.

Exemplul 3.2.27. Considerăm clasa C1, care conține variabilele i, j ce sunt incrementate în metoda f(). Metoda afisare() va tipări valoarea acestora. Clasa C2 extinde clasa Thread (referitoare la fire de execuție, vom reveni în Capitolul 5) și conține în metoda run() o structură de tip for care va realiza 100 de iterații. Clasa C3 creează în metoda main() două fire de execuție care vor apela metoda f(), respectiv afisare().

```
// declararea clasei C1
// aceasta contine doar membri statici
class C1 {
    static int i = 0, j = 0;
    static void f() {
        i++;
        j++;
    }
    static void afisare() {
        System.out.print("i = " + i);
        System.out.println(" j = " + j);
    }
}

// se declara clasa corespunzatoare unui fir de executie
class C2 extends Thread {
    private C1 c1;
    private String numeFir;

    public C2(C1 c1, String numeFir) {
        this.c1 = c1;
        this.numeFir = numeFir;
    }

    // metoda run() se executa la pornirea firului de executie
    public void run() {
        for (int i = 1; i <= 100; i++) {
            System.out.println(numeFir + " este la pasul " + i);
            if (numeFir.equals("FirUnu")) c1.f();
            else c1.afisare();
        }
    }
}
```

```
public class C3 {
    public static void main(String args[]) {
        C1 obiect = new C1();
        // se creeaza doua fire de executie
        C2 fir1 = new C2(obiect, "FirUnu");
        C2 fir2 = new C2(obiect, "FirDoi");
        // se pornesc firele de executie
        // metoda start() se termina imediat
        fir1.start();
        fir2.start();
    }
}
```

La execuția acestui program, pot interveni situații când metoda afiseaza() va tipări o valoare mai mare pentru j decât valoarea lui i, deoarece exemplul nostru nu conține mecanisme de sincronizare. Astfel, se poate afișa ceva de genul:

```
FirDoi este la pasul 8
i=37 j=37
FirDoi este la pasul 9
i=37FirUnu este la pasul 39
j=38
FirDoi este la pasul 10
i=38 j=38
```

O primă modalitate de prevenire a acestui lucru este adăugarea în lista modificatorilor metodelor f() și afiseaza() a cuvântului rezervat synchronized (vom reveni în Capitolul 6). Aceasta va preveni ca metodele f() și afiseaza() să poată fi apelate de cel mult un fir de execuție (ne-concurent) și este clar că valorile divizate ale lui i și j vor fi actualizate înaintea returnării din metoda f(). Așadar, valoarea lui j nu va fi niciodată mai mare decât cea a lui i în cadrul metodei afiseaza().

A doua modalitate este să declarăm atributele x și y din clasa C1 ca fiind volatile. Aceasta va permite ca metodele f() și afiseaza() să fie executate concurent, dar garantează accesarea valorilor divizate pentru i și j exact în același timp și în aceeași ordine cum apar în program. Astfel, metoda afiseaza() nu va afișa valoarea lui j mai mare decât a lui i.

Continuăm acum cu inițializarea atributelor, care în principiu au semnificația unei instrucțiuni de asignare atașate unei variabile declarate, astfel:

1. Dacă declarația se referă la o variabilă clasă (adică atribut static), atunci inițializatorul variabilei este evaluat și se face o singură dată asignarea atunci când este inițializată clasa;
2. Dacă declarația se referă la o variabilă instanță (adică atribut nestatic), atunci inițializatorul variabilei este evaluat și se face asignarea de fiecare dată când se creează o instanță a clasei.

În ceea ce privește o expresie de inițializare a unei variabile clasă, aceasta nu trebuie să utilizeze variabile a căror valoare nu se cunoaște sau chiar variabila însăși.

Exemplul 3.2.28. Fie clasa de mai jos:

```
class EroareInitializareVariabileClasa {
    static int a = b; // initializarea 1
    static int b = 3; // initializarea 2
    static int c = c + 3; // initializarea 3
}
```

Inițializarea 1 este greșită (și va conduce la eroare la compilare), deoarece variabila *b* nu are valoarea cunoscută la momentul inițializării lui *a*. Inițializarea 2 este corectă, în schimb inițializarea 3 este greșită (și va conduce la eroare la compilare), deoarece variabila *c* nu are valoarea cunoscută la momentul inițializării sale.

Dacă o referință a unui nume simplu al oricărei variabile instanță sau cuvintele rezervate *this* și *super* apar într-o expresie de inițializare pentru o variabilă clasă, atunci apare o eroare la compilare. Explicația se datorează faptului că atributele statice se inițializează în momentul încărcării claselor, deci înaintea creării de instanțe pentru acestea.

Ca și în cazul variabilelor clasă, în ceea ce privește o expresie de inițializare a unei variabile instanță, aceasta nu trebuie să folosească variabile a căror valoare nu se cunoaște sau chiar variabila însăși. În schimb, o expresie de inițializare a unei variabile instanță poate conține un nume de variabilă statică, precum și cuvintele rezervate *this* și *super*.

Exemplul 3.2.29. Fie programul Java de mai jos:

```
class C1 {
    int a = b + 1; // initializarea 1
    static int b = 3; // initializarea 2
    int c = this.a + 5; // initializarea 3
}

public class InitializareVariabileInstanta {
    public static void main(String args[]) {
        C1 obiect = new C1();
        System.out.println("a = " + obiect.a + ", b = " +
            obiect.b + ", c = " + obiect.c);
    }
}
```

Inițializarea 1 este corectă deoarece variabila *b* are valoarea cunoscută la momentul inițializării lui *a*. Evident, inițializarea 2 este corectă. Inițializarea 3 este corectă deoarece variabila *a* are valoarea cunoscută la momentul inițializării atributului *c*. Execuția programului Java de mai sus va afișa:

```
a = 4, b = 3, c = 9
```

Continuăm cu prezentarea unor exemple de declarații de atribute. Referirea unui atribut static dintr-o clasă de bază se poate realiza fie prin cuvântul rezervat *super*, fie prin numele clasei.

Exemplul 3.2.30. Fie programul Java de mai jos:

```
class C1 {
    static int x = 4;
}

class SupraScriereAtributeStatice extends C1 {
    static double x = 2.3;
    public static void main(String[] args) {
        new SupraScriereAtributeStatice().afisare();
    }
    void afisare() {
        System.out.println(x + " " + super.x);
        System.out.println(SupraScriereAtributeStatice.x + " " +
            C1.x);
    }
}
```

Atributul static *x* apare atât în clasa *SupraScriereAtributeStatice*, cât și în clasa de bază *C1*. Astfel, atributul *x* din clasa *C1* este suprascris (sau ascuns) de către cel din clasa *SupraScriereAtributeStatice*. Programul va afișa la execuție:

```
2.3 4
2.3 4
```

În ceea ce privește suprascrierea (ascunderea) atributelor instanță, există (cel puțin) trei modalități de accesare a acestora:

1. utilizând cuvintele rezervate *this* (care se referă la obiectul curent) și *super* (care se referă la atributele din clasa de bază a obiectului curent);
2. utilizând o conversie explicită la tipul de bază;
3. creând câte un obiect pentru fiecare clasă în parte.

Exemplul 3.2.31. Programul Java de mai jos pune în evidență toate cele trei situații discutate:

```
class C1 {
    int x = 4;
}

class SupraScriereAtributeInstanta extends C1 {
    double x = 2.3;
    public static void main(String[] args) {
        SupraScriereAtributeInstanta obiectUnu =
```

```

        new SupraScriereAtributeInstanta();
    obiectUnu.afisare();
    System.out.println(obiectUnu.x + " " + ((C1)obiectUnu).x);
    C1 obiectDoi = new C1();
    System.out.println(obiectUnu.x + " " + obiectDoi.x);
}
void afisare() {
    System.out.println(this.x + " " + super.x);
}
}

```

Se va afișa la execuție:

```

2.3 4
2.3 4
2.3 4

```

O clasă poate moșteni două sau mai multe atribute cu același nume de la două interfețe sau de la o superclasă și o interfață. Pentru a evita erorile de compilare care pot apărea din cauza ambiguității numelui atributului, se poate utiliza un nume calificat sau o expresie de acces care folosește cuvântul rezervat `super`.

Exemplul 3.2.32. Programul Java de mai jos pune în evidență rezolvarea ambiguității unui atribut cu același nume dintr-o clasă, respectiv o interfață:

```

interface I1 { float x = 2.3f; }
class C1 { int x = 4; }

public class SuprascriereMultipla extends C1 implements I1 {
    public static void main(String[] args) {
        new SuprascriereMultipla().afisare();
    }
    void afisare() {
        System.out.println(super.x + " " + I1.x);
    }
}

```

Expresia de acces `super.x` se va referi la atributul `x` din clasa `C1`, iar numele calificat `I1.x` se va referi la atributul `x` al interfeței `I1`. La execuție se va afișa:

```

4 2.3

```

Dacă un atribut este moștenit prin mai multe căi dintr-o interfață, acesta se consideră că este moștenit o singură dată și se poate accesa prin numele său simplu fără ambiguitate.

Exemplul 3.2.33. Programul Java de mai jos pune în evidență rezolvarea ambiguității unui atribut cu același nume moștenit de două ori dintr-o interfață:

```

interface I1 { int x = 2; }

```

```

interface I2 extends I1 { }
class C1 { }
class C2 extends C1 implements I1 { }

public class C3 extends C2 implements I2
{
    public static void main(String args[]) {
        System.out.println("x = " + x);
    }
}

```

Atributul `x` este moștenit și din clasa `C2` (prin implementarea interfeței `I1`), cât și din interfața `I2` (prin derivarea din `I1`). Programul va afișa la execuție:

```

x = 2

```

3.2.6.3. Declarațiile metodelor unei clase

O metodă declară cod executabil care poate fi executat trimițând un număr fix de valori ca argumente. <DeclarațieMetoda> (din definiția declarării unei clase, secțiunea 3.2) are sintaxa generală:

```

[<ModificatorMetoda>] <TipRezultat> <Identificator>
([<ListaParametriFormali>])
throws <TipClasa1> [, . . . , <TipClasaN>] <CorpMetoda>

```

unde:

1. <ModificatorMetoda> poate fi: `public`, `protected`, `private`, `abstract`, `final`, `static`, `synchronized`, `native` (fiecare modificator poate apărea cel mult o dată, iar dintre primii trei modificatori poate apărea cel mult unul; în caz contrar, se obține eroare la compilare);
2. <TipRezultat> poate fi `void` sau un tip Java (primitiv sau referință);
3. <Identificator> este numele metodei;
4. <ListaParametriFormali> poate lipsi, dar dacă este prezentă, atunci are forma:

```

<Tip1> <Identificator1> []...[], ..., <TipN> <IdentificatorN>
[]...[],

```

unde identificatorii reprezintă parametri formali, iar dacă se referă la variabile de tip tablou, atunci vor conține paranteze pătrate `[]`, altfel acestea vor lipsi;
5. <CorpMetoda> poate fi un bloc de metodă sau nimic și atunci este specificat prin `" ; "`.

Numele metodei (<Identificator>) poate fi folosit pentru referirea către acea metodă și poate coincide cu numele clasei sau cu un atribut al clasei. Dacă există deja definită o metodă cu aceeași *signatură* (nume, număr de parametri și tipurile

parametrilor), atunci se obține o eroare la compilare. Metodele și atributele pot avea același nume, pentru că se utilizează în contexte diferite.

Parametrul formal este cel care apare în definiția (prototipul) unei metode. La apelul acesteia, parametrii sunt actuali, iar valorile acestora sunt transmise parametrilor formali.

Nu este posibil ca doi parametri formali să aibă același nume (se obține eroare la compilare). Când se apelează o metodă, valorile expresiilor parametrilor actuali vor inițializa variabilele parametru nou create. Domeniul de vizibilitate a numelor parametrilor formali este întreg corpul metodei. Aceste nume de parametri nu pot fi redeclarat ca variabile locale sau parametri excepție în metodă (deci nu este permisă ascunderea numelor parametrilor).

Exemplul 3.2.34. Următoarea clasă va produce o eroare la compilare, deoarece metoda `f()` nu poate fi supraîncărcată (cele două definiții ale lui `f()` au aceeași semnătură, diferă doar tipul întors):

```
class C1 {
    int x;
    int f(int x) { return x; }
    void f(int x) { x = this.x; }
}
```

Continuăm cu prezentarea modificatorilor metodelor. Dacă există mai mulți modificatori, atunci se preferă ordinea menționată în definiția de mai sus.

Începem prezentarea modificatorului `abstract`. O declarație a unei metode abstracte specifică faptul că acea metodă este membru al clasei, furnizându-i semnătura, tipul returnat și clauzele `throws`, dar fără a descrie o implementare a sa. Dacă o metodă este declarată `abstract`, atunci și clasa din care face parte trebuie declarată `abstract`, altfel apare o eroare la compilare. Fiecare subclasă care nu este declarată `abstract` trebuie să furnizeze o implementare a metodei abstracte, altfel apare o eroare la compilare.

Dacă o metodă conține modificatorul `abstract`, atunci nu mai poate conține modificatorii: `private`, `static`, `final`, `native` sau `synchronized`. Este imposibil să se implementeze o metodă `private abstract`, deoarece metodele `private` nu sunt accesibile în subclase. De asemenea, este imposibil să se implementeze o metodă `abstract final`, deoarece metodele `final` nu mai pot fi suprascrise. Este imposibil să se implementeze o metodă `abstract static`, deoarece metodele `static` trebuie să fie implementate la încărcarea claselor.

Exemplul 3.2.35. O metodă instanță care nu este declarată `abstract` poate fi suprascrisă de o metodă declarată `abstract`.

```
class C1 {
    public String f() {
        // se returneaza numele clasei
        return this.getClass().getName();
    }
}
```

```
}
}

// definirea unei clase abstracte
abstract class C2 extends C1 {
    int x = 2;
    // supradefinirea metodei f() mostenita din C1
    public abstract String f();
    protected String g() {
        // se apelează metoda f() din clasa C1
        return super.f() + ", x = " + x;
    }
}

// se declara clasa C3 care nu este abstracta
class C3 extends C2 {
    int y = 3;
    // se redefineste metoda f() mostenita din C2
    public String f() {
        return g() + ", y = " + y;
    }
}

public class TestAbstract {
    public static void main(String args[]) {
        C3 object = new C3();
        // se apelează metoda f() din clasa C3
        System.out.println(object.f());
    }
}
```

Metoda `f()` din clasa `C2` suprascrive o metodă deja implementată în clasa `C1`. Metoda `g()` din clasa `C2` apelează metoda `f()` din clasa `C1` al cărei cod există. În metoda `g()` este esențială prezența cuvântului rezervat `super`, în caz contrar s-ar fi obținut eroare la compilare, deoarece s-ar fi apelat metoda `f()` a clasei `C2`, al cărei cod lipsea. Clasa `C3` trebuie să conțină definiția metodei `f()`, în caz contrar s-ar fi obținut eroare la compilare, pentru că `C3` nu a fost declarată `abstract`. În clasa de aplicație `TestAbstract`, se apelează pentru obiect din clasa `C3` metoda `f()`, care la rândul ei apelează metoda `g()` din clasa `C2` și care apelează în sfârșit metoda `f()` din clasa `C1`. Astfel, programul Java de mai sus va afișa:

```
C3, x = 2, y = 3
```

O metodă declarată `final` nu poate fi suprascrisă (adică redefinită într-o clasă derivată) sau ascunsă (declarată `private`), în caz contrar se obține eroare la compilare. O metodă declarată `private` și toate metodele declarate într-o clasă `final` sunt

implicit finale, deoarece este imposibilă suprascrierea acestora. De asemenea, este o eroare la compilare ca o metodă să fie declarată concomitent final și abstract.

O metodă declarată static se numește *metodă a clasei* și poate fi apelată fără referință la un obiect particular. O încercare de referire la obiectul curent folosind this sau super în corpul metodei clasei conduce la o eroare de compilare. O metodă care nu este declarată static se numește *metodă instanță* (sau metodă ne-statică). O metodă instanță se apelează în raport cu un obiect, numit *obiect curent*, care poate fi referit cu this sau super. O metodă statică nu poate fi declarată abstract, în caz contrar apare o eroare la compilare.

O metodă synchronized (sincronizată) trebuie să capete dreptul de exclusivitate înainte de execuția acesteia, în sensul că nici un alt fir de execuție nu va executa în acel moment respectiva metodă. Detalii asupra metodelor sincronizate vor fi descrise în Capitolul 6 (Fire de execuție).

Clasa java.lang.Math conține metoda:

```
public static synchronized double random();
```

care returnează un număr aleator de tip double în „intervalul” [0.0, 1.0). Această metodă este declarată synchronized pentru a permite utilizarea sa corectă de mai multe fire de execuție (astfel încât calculul următorului număr pseudo-aleator să fie realizat corect).

Continuăm cu prezentarea metodelor declarate native. O metodă Java este *nativă* dacă aceasta este definită în alt limbaj de programare (de exemplu, C, C++) și este doar declarată într-un program Java de unde este apelată. O metodă declarată native nu poate fi declarată și abstract, deoarece codul acesteia se definește în biblioteca dinamică, și nu într-o subclasă. Una din trăsăturile importante ale limbajului Java este independența de platformă, adică codul compilat pe o platformă poate să ruleze pe orice altă platformă care are un interpretor Java. Folosind metode native, acest avantaj se pierde, deoarece aplicațiile vor rula doar pe platformele aceluiasi sistem de operare și folosind aceleași biblioteci dinamice capabile să memoreze metodele native ale aplicației. Începând cu Java versiunea 1.1, a fost creată *interfața nativă Java* (eng. *Java Native Interface* – JNI) care permite apelul metodelor native dintr-o aplicație Java.

Exemplul 3.2.36. În cele ce urmează, vom prezenta o aplicație simplă Java care apelează o metodă nativă scrisă în C, care la rândul ei apelează funcția MessageBox() din Win32 API (eng. *Windows 32 Application Programming Interface*) pentru a afișa o cutie grafică care conține un text. Sub platforma Windows trebuie urmați cinci pași:

1. Scrierea programului Java (numit AfisareMesajJNI.java) care declară metoda nativă C:

```
public class AfisareMesajJNI {
    public static void main(String [] args) {
        AfisareMesajJNI obiect = new AfisareMesajJNI();
        obiect.afisareMesaj("Text afisat folosind JNI");
    }
}
```

```
private native void afisareMesaj(String mesaj);
static {
    System.loadLibrary("testAfisare");
}
}
```

Metoda nativă afisareMesaj() va fi definită în biblioteca dinamică testDoi.dll.

2. Compilarea programului de mai sus folosind comanda:

```
javac AfisareMesajJNI
```

care conduce la obținerea fișierului AfisareMesajJNI.class.

3. Crearea automată a fișierului interfață asociat fișierului AfisareMesajJNI.class folosind comanda:

```
javah -jni AfisareMesajJNI
```

Această comandă va genera pentru fiecare declarație de metodă native câte un prototip de funcție într-un fișier de interfață C, numit AfisareMesajJNI.h.

4. Scrierea aplicației C care va genera *biblioteca de legături dinamice* (eng. *Dynamic Link Library* – DLL). Aplicația de mai jos include fișierul de interfață AfisareMesajJNI.h și implementează metoda nativă afisareMesaj(). Deoarece această parte este dependentă de platformă, am scris aplicația în C Win32 API. Dacă se utilizează altă platformă, pentru a afișa mesajul se înlocuiește <windows.h> cu <stdio.h> și apelul funcției MessageBox() cu printf().

```
#include <windows.h>
#include "AfisareMesajJNI.h"

BOOL APIENTRY DllMain(HANDLE hModule,
    DWORD dwReason, void ** lpReserved) {
    return TRUE;
}

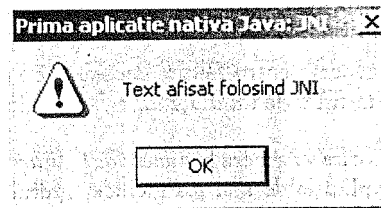
JNIEXPORT void JNICALL Java_AfisareMesajJNI_afisareMesaj(JNIEnv
    * jEnv, jobject this, jstring jMsg) {
    const char * msg;
    msg = (*jEnv) -> GetStringUTFChars(jEnv, jMsg, 0);
    MessageBox(HWND_DESKTOP, msg,
        "Prima aplicatie nativa Java: JNI",
        MB_OK | MB_ICONEXCLAMATION);
    (*jEnv) -> ReleaseStringUTFChars(jEnv, jMsg, msg);
}
```

Folosind mediul Microsoft Visual C++ (de exemplu, versiunea 6.0), crearea unei biblioteci de legături dinamice se face după următorii pași:

- Se selectează (în ordine) File, New, apoi Projects;
- Din lista de aplicații se selectează Win32 Dynamic-Link Library, iar apoi se completează numele proiectului (eng. *Project Name*) cu testAfisare (trebuie să coincidă cu numele argumentului metodei `loadLibrary()` din programul Java de la pasul 1);
- Se adaugă la proiectul creat fișierul C de mai sus, după care se compilează întreg proiectul.

Astfel s-a creat în subdirectorul Debug fișierul `testAfisare.dll`.

5. Execuția aplicației Java necesită existența bibliotecii `testAfisare.dll` în directorul în care se află fișierul `AfisareMesajJNI.class`. Lansând comanda `java AfisareMesajJNI`, obținem o fereastră de dialog care conține textul dorit:



Continuăm cu prezentarea clauzei `throws` care apare în declarațiile metodelor unei clase. Aceasta a mai fost prezentată succint în Capitolul 2, Secțiunea 2.9.11, Exemplele 2.9.12, 2.9.13, 2.9.15, 2.9.17, când s-a descris instrucțiunea `throw`. O clauză `throws` este utilă la declararea oricăror excepții care pot rezulta în urma execuției unei metode sau constructor. Așa cum s-a menționat și în definiția unei metode, sintaxa generală a definiției clauzei `throws` este:

```
throws <TipClasa1> [, . . . , <TipClasaN>]
```

unde `<TipClasa1>`, ..., `<TipClasaN>` sunt clase `Throwable` sau subclase ale clasei `Throwable`, în caz contrar se obține o eroare la compilare.

Corpul unei metode va fi prezentat în continuare. Acesta poate fi bloc de instrucțiuni care implementează metoda sau doar indicând lipsa implementării. Corpul unei metode este ; dacă și numai dacă metoda este declarată `abstract` sau `native`. Dacă metoda este declarată `abstract` sau `native` și este definită printr-un bloc de instrucțiuni, atunci se obține eroare la compilare. Și reciproc, dacă o metodă nu este declarată `abstract` sau `native` și nu este definită, atunci se obține eroare la compilare. Dacă o metodă este declarată `void`, atunci corpul său nu trebuie să conțină o instrucțiune `return` cu expresie, în caz contrar apare o eroare de compilare. O metodă care are tip returnat trebuie să încheie execuția metodei printr-o instrucțiune `return` cu expresie sau printr-o instrucțiune `throw`.

Exemplul 3.2.37. Programul Java de mai jos scoate în evidență o situație când nu toate ramurile unei instrucțiuni `if` au asociată o instrucțiune (obligatorie) `return`:

```
class C1 {
    int f(int x) {
        if (x < 10) return x;
        else throw new RuntimeException("x este prea mare!");
    }

    public class TestReturn {
        public static void main(String args[]) {
            C1 obiect = new C1();
            System.out.println(obiect.f(9));
            System.out.println(obiect.f(12));
        }
    }
}
```

Acesta va afișa la execuție:

```
9
java.lang.RuntimeException:
x este prea mare!
    at C1.f(TestReturn.java:4)
    at TestReturn.main(TestReturn.java:11)
Exception in thread "main"
```

Continuăm cu noțiunile de suprascriere, ascundere și supraîncărcare a metodelor unei clase. O clasă moștenește de la superclasa directă și de la superinterfețele directe toate metodele (chiar dacă sunt declarate `abstract`) accesibile din codul clasei și care nu sunt suprascrise și nu sunt ascunse de declarația clasei. Dacă o clasă declară o metodă instanță, spunem că declarația metodei *suprascrie* toate metodele cu aceeași semnătură din superclasele și superinterfețele clasei (care altfel ar fi fost accesibile codului clasei). În plus, dacă declarația metodei din clasă nu este `abstract`, atunci spunem că declarația metodei *implementează* toate metodele cu aceeași semnătură din superclasele și superinterfețele clasei (care altfel ar fi fost accesibile codului clasei).

O metodă instanță nu poate suprascrie o metodă declarată `static` (din acest punct de vedere, suprascrierea metodelor diferă de ascunderea atributelor, unde era permis pentru o variabilă instanță să ascundă o variabilă `static`), în caz contrar apare o eroare la compilare.

O metodă suprascrisă poate fi accesată prin utilizarea unei expresii care conține cuvântul rezervat `super`. Spre deosebire de accesul la un atribut ascuns, apelul unei metode suprascrise nu se poate face prin nume calificat sau printr-o conversie la tipul superclasei.

Exemplul 3.2.38. Programul Java de mai jos pune în evidență o situație de suprascriere a unei metode instanță. Astfel, se va apela metoda `f()` a clasei `C1` la fiecare apel al lui `f()` al clasei `C2`, chiar dacă tipul obiectului curent este `C1`.

```
class C1 {
```

```

int x = 1;
void f(int x) {
    this.x = x;
    System.out.println("Metoda f() din C1");
}

class C2 extends C1 {
    int xDoi = 5;
    void f(int x) {
        System.out.println("Metoda f() din C2");
        super.f(g(x));
        System.out.println("Metoda f() din C2");
    }
    int g(int x) {
        return x > xDoi ? x : xDoi;
    }
}

public class Suprascriere {
    public static void main(String args[]) {
        C1 obiect = new C2();
        obiect.f(4);
        System.out.println(obiect.x);
    }
}

```

Acest program va afișa la execuție:

```

Metoda f() din C2
Metoda f() din C1
Metoda f() din C2
5

```

Dacă o clasă declară o metodă statică, atunci spunem că declarația metodei *ascunde* toate metodele cu aceeași semnătură din superclasele și superinterfețele (care altfel ar fi fost accesibile codului clasei). Dacă o metodă statică ascunde o metodă instanță, atunci apare o eroare la compilare (spre deosebire de ascunderea unui atribut care este permisă). O metodă ascunsă poate fi accesată prin nume calificat, printr-o expresie care conține cuvântul rezervat `super` sau cu o conversie la un tip superclasă (la fel ca la ascunderea atributelor).

Dacă o declarație de metodă suprascrie sau ascunde declarația altei metode, atunci acestea trebuie să returneze același tip. Modificatorul de acces al unei metode suprascrise și ascunse trebuie să furnizeze cel puțin la fel de mult acces ca al metodei suprascrise, în caz contrar apare o eroare de compilare, astfel:

- dacă metoda suprascrisă sau ascunsă este `public`, atunci metoda care o suprascrie sau o ascunde trebuie să fie declarată `public`;

- dacă metoda suprascrisă sau ascunsă este `protected`, atunci metoda care o suprascrie sau o ascunde trebuie să fie `public` sau `protected`;
- dacă metoda suprascrisă sau ascunsă are acces implicit (fără a preciza nici un modificator), atunci metoda care o suprascrie sau o ascunde trebuie să nu fie `private`.

O metodă declarată `private` nu este accesibilă subclaselor și deci nu poate fi vorba despre suprascrierea sau ascunderea ei.

Exemplul 3.2.39. Programul de mai jos exemplifică ascunderea atributelor și suprascrierea metodelor.

```

class C1 {
    int x = 1;
    void f(int x) { this.x = x; }
    int getX_C1() { return x; }
}

class C2 extends C1 {
    float x = 5.0f;
    void f(int x) { super.f((int)x); }
    float getX_C2() { return x; }
}

public class SuprascriereSiAscundere {
    public static void main(String args[]) {
        C2 obiect = new C2();
        obiect.f(4);
        System.out.println(obiect.getX_C2());
        System.out.println(obiect.getX_C1());
    }
}

```

Clasa C2 ascunde declarația variabilei instanță `x` a clasei C1 cu variabila instanță `x` a clasei C1, apoi suprascrie metoda `f()` a clasei C1 cu propria metodă `f()`. La apelul `obiect.f(4)` se va modifica atributul `x` al clasei C1 la valoarea 4. Execuția aplicației va afișa:

```

5.0
4

```

Este posibil pentru o clasă să moștenească mai mult de o metodă cu aceeași semnătură, astfel:

1. Dacă una dintre metodele moștenite nu este abstractă, atunci există subcazurile:
 - a) Dacă metoda care nu este abstractă este statică, atunci se obține eroare la compilare;

- b) Altfel, metoda care nu este abstractă se consideră că suprascrie și deci implementează toate celelalte metode pe care clasa le moștenește. Dacă tipul returnat al metodei care suprascrie este altfel decât al metodelor suprascrise, atunci se obține eroare la compilare. Același lucru se obține dacă metoda care nu este abstractă are o clauză throws, care are conflicte cu altă metodă moștenită.
2. Dacă toate metodele moștenite sunt abstracte, atunci clasa este abstractă și moștenește toate metodele abstracte. Metodele moștenite (suprascrise) trebuie să aibă același tip returnat.

Exemplul 3.2.40. Programul de mai jos pune în evidență faptul că la suprascrierea metodelor trebuie să coincidă și lista excepțiilor clauzei throws, care pot fi aruncate în timpul apelului metodei:

```
class ExceptiaNoastra extends Exception {
    ExceptiaNoastra() { super(); }
}

class C1 {
    int x;
    void f(int x) { this.x += x; }
}

public class ClasaDerivata extends C1 {
    void f(int x) throws ExceptiaNoastra {
        if (x < 0) throw new ExceptiaNoastra();
        this.x = x;
    }
}
```

Programul va furniza eroare la compilare, deoarece metoda suprascrisă `f()` din `C1` nu are specificată nici o clauză throws așa cum are metoda `f()` din `ClasaDerivata`. Evident, programul devine corect dacă adăugăm în antetul metodei `f()` din clasa `C1` clauza throws `ExceptiaNoastra`.

O metodă clasă ascunsă (statică) poate fi invocată folosind o referință al cărei tip este clasa care conține declarația metodei. În concluzie, ascunderea metodelor statice diferă de suprascrierea metodelor instanță.

Exemplul 3.2.41. Programul de mai jos evidențiază diferența dintre ascunderea metodelor statice și suprascrierea metodelor instanță.

```
class C1 {
    static String f() { return "Mesajul Unu din C1"; }
    String g() { return "Mesajul Doi din C1"; }
}
```

```
class C2 extends C1 {
    static String f() { return "Mesajul Unu din C2"; }
    String g() { return "Mesajul Doi din C2"; }
}

public class DiferentaStaticInstanta {
    public static void main(String[] args) {
        C1 obiect = new C2();
        System.out.println(obiect.f() + ", " + obiect.g());
    }
}
```

Obiectul creat aparține tipului clasă `C1` și deci apelul `obiect.f()` se va referi la metoda statică `f()` din clasa `C1`. În schimb, apelul `obiect.g()` se va referi la metoda `g()` din clasa `C2`, deoarece aceasta suprascrie metoda `g()` din clasa `C1`. Astfel, execuția va afișa:

```
Mesajul Unu din C1, Mesajul Doi din C2
```

Continuăm cu supraîncărcarea metodelor unei clase. Dacă două metode ale unei clase (declarate în aceeași clasă sau moștenite de o clasă sau una declarată și una moștenită) au același nume, dar semnături diferite, atunci se spune că numele metodei este *supraîncărcat* (se mai spune că metoda este *supraîncărcată*). Metodele sunt suprascrise comparându-se semnăturile. Dacă avem două metode declarate într-o clasă `C1`, să zicem `public void f();` și `public void f(int);`, și subclasa `C2` suprascrie una din ele, de exemplu `public void f();`, atunci subclasa `C2` va moșteni metoda cealaltă, adică `public void f(int);` (din acest punct de vedere, Java diferă de C++). În cazul metodelor supraîncărcate, numărul de parametri actuali și tipurile argumentelor din momentul compilării vor determina semnătura metodei care va fi apelată. Dacă metoda care trebuie apelată este metodă instanță, metoda actuală care trebuie să fie apelată va fi determinată în momentul execuției, folosind metode dinamice.

Exemplul 3.2.42. Se evidențiază supraîncărcarea metodelor unei clase:

```
class C1 {
    double x = 5.0f;
    void f(int x) {
        f((double)x + 2);
    }
    void f(double x) {
        this.x = x;
    }
}
```

```

public class Supraincarcare {
    public static void main(String args[]) {
        Cl obiect = new Cl();
        obiect.f(4.0);
        System.out.println(obiect.x);
    }
}

```

Fiind metodă instanță, la momentul execuției se va decide care metodă `f()` se va apela. Cum literalul `4.0` este de tip `double`, evident se va apela `f(double)` și programul va afișa la execuție:

```
4.0
```

3.2.7. Inițializatori statici

Inițializatorii statici declarați într-o clasă sunt executați când clasa este inițializată și împreună cu inițializatorii atributelor pentru variabilele clasă pot fi utilizați pentru inițializarea variabilelor clasă. Sintaxa generală a inițializatorilor statici este:

```
static <Bloc>
```

Inițializatorii statici și inițializatorii variabilelor clasă sunt executați în ordinea în care apar și nu se pot referi la variabilele clasă declarate după utilizarea lor, chiar dacă aceste variabile de clasă sunt în domeniu. Această restricție are ca scop determinarea la momentul compilării a inițializărilor circulare. Într-un inițializator static nu poate apărea instrucțiunea `return` sau cuvintele rezervate `this` și `super`, în caz contrar se semnalează o eroare la compilare.

Exemplul 3.2.43. Următorul program Java justifică importanța ordinii inițializatorilor statici:

```

class Cl {
    static int f() { return j; }
    static int i = f();
    static int j = 1;
}

public class TestStatic {
    public static void main(String[] args) {
        System.out.println(Cl.i);
    }
}

```

Inițializatorul variabilei `i` folosește metoda `f()` pentru a accesa valoarea variabilei `j` înainte ca `j` să fie inițializat cu 1. Astfel, execuția programului de mai sus va afișa:

```
0
```

3.2.8. Declarațiile constructorilor

Un *constructor* este utilizat la crearea unui obiect care este instanță a unei clase. Declarația unui constructor arată ca declarația unei metode, dar care nu are tip rezultat (nu returnează nimic, nici măcar `void`), astfel:

```

[public | protected | private] <NumeClasa>
([<ListaParametriFormali>])
[throws <ListaClase>] {
    [<ApelExplicitConstructor>]
    [<BlocInstrucțiuni>]
}

```

unde:

1. `<NumeClasa>` reprezintă numele clasei în care este declarat constructorul;
2. `<ListaParametriFormali>` are aceeași sintaxă și semantică ca la declarația metodelor;
3. clauza opțională `throws <ListaClase>` are aceeași sintaxă și semantică ca la declarația metodelor;
4. `<ApelExplicitConstructor>` poate fi de forma:
 - a) `this(<ListaArgumente>);`
 - b) `super(<ListaArgumente>);`

Constructorii sunt apelați de expresii de creare a instanței clasei, de apelul metodei `newInstance()` a clasei `Class`, de conversiile și concatenările rezultate în urma operatorului de concatenare `+` și de apelul explicit al acestora din alți constructori. Constructorii nu pot fi apelați în expresii de apel al unei metode. Accesul la constructori se realizează după modificatorii de acces. Constructorii nu sunt membri, deci nu pot fi niciodată subiect de ascundere sau suprascriere. Signatura unui constructor este exact aceeași cu structura și comportarea semnăturii unei metode.

Spre deosebire de metode, un constructor nu poate fi `final`, `abstract`, `static`, `synchronized` sau `native`. Un constructor nu se moștenește, deci nu are sens să poată fi declarat `final` sau `abstract`. Un constructor se apelează relativ la un obiect, deci nu are sens să fie `static`. Nu există o necesitate practică pentru ca un constructor să fie `synchronized`, deoarece acesta va bloca obiectul care se construiește, care oricum nu este accesibil altor fire de execuție.

O clasă poate interzice crearea de instanțe ale clasei prin declararea a cel puțin unui constructor (pentru a preveni crearea unui constructor implicit) și declarând toți constructorii `private`.

Exemplul 3.2.44. Următorul program Java pune în evidență situația când o clasă nu poate avea instanțe:

```

class Cl {
    int x = 1;
}

```

```
private C1() {}
}
```

Clasa C1 nu poate avea instanțe, încercarea de a crea obiecte din clasa C1 conduce la o eroare la compilare. Putem permite crearea de obiecte doar în același pachet prin acces implicit la constructor (fără modificatori), în rest nefiind posibilă crearea de obiecte:

```
package P1;

public class C1 {
    int x = 1;
    C1() {}
}
```

Exemplul 3.2.45. Următorul program Java continuă Exemplul 3.2.41 în sensul „controlării” mai exacte a numărului de instanțe pe care le poate avea o clasă:

```
class ExceptiaNoastra extends Exception {
    ExceptiaNoastra(String s) { super(s); }
}

class C1 {
    static int numarInstante = 0;
    static int numarMaximInstante = 3;
    int x = 1;
    C1(int x) throws ExceptiaNoastra {
        numarInstante++;
        if (numarInstante > numarMaximInstante)
            throw new ExceptiaNoastra(
                "Am depasit numarul de instante");
        System.out.println("Obiect " + numarInstante);
        this.x = x;
    }
}

public class NumarMaximDeInstante {
    public static void main(String args[]) {
        try {
            C1 obiectUnu = new C1(1);
            C1 obiectDoi = new C1(2);
            C1 obiectTrei = new C1(3);
            C1 obiectPatru = new C1(4);
        }
        catch (ExceptiaNoastra e) {
```

```
System.out.println("Exceptie raportata: " + e);
}
```

Execuția acestui program conduce la afișarea:

```
Obiect 1
Obiect 2
Obiect 3
Exceptie raportata: ExceptiaNoastra: Am depasit numarul de
instante
```

Prima instrucțiune a corpului unui constructor poate fi apelul explicit al altui constructor al aceleiași clase cu ajutorul cuvântului rezervat `this`, urmat de o listă de argumente, sau apelul explicit al unui constructor al unei superclase directe, utilizând cuvântul rezervat `super` urmat de o listă de argumente între paranteze. Apelul explicit al unui alt constructor folosind `this` sau `super` trebuie să fie prima instrucțiune, altfel se obține eroare la compilare. Astfel, apelarea directă sau indirectă a unui constructor cel puțin o dată conduce la eroare de compilare. În cazul în care corpul unui constructor nu începe cu apelul explicit al altui constructor și constructorul declarat nu este parte a clasei `Object`, atunci se presupune implicit de către compilator că se începe corpul constructorului cu un apel al constructorului superclasei directe fără argumente (adică de forma `super()`). Cu excepția apelurilor explicite ale constructorilor, corpul unui constructor este similar corpului unei metode, putând conține chiar instrucțiuni `return`, însă fără expresie atașată. Variabilele instanță (adică atributele ne-stactice), metodele instanță declarate într-o clasă sau într-o superclasă, cuvintele rezervate `this` și `super` nu pot fi folosite în apelul unui constructor explicit, în caz contrar apare o eroare la compilare.

Exemplul 3.2.46. Următorul program Java evidențiază apelurile explicite ale constructorilor prin intermediul cuvintelor rezervate `this` și `super`:

```
class C1 {
    int x = 1;
    C1(int x) { this.x = x; }
}

class C2 extends C1 {
    static final int y = 2;
    int z = 3;
    C2(int x) {
        this(x, y);
    }
    C2(int x, int z) {
        super(x);
```

```

        this.z = z;
    }
    public String toString() {
        return "x = " + x + ", y = " + y + ", z = " + z;
    }
}

public class ThisSuper {
    public static void main(String args[]) {
        C2 obiectUnu = new C2(4);
        C2 obiectDoi = new C2(5, 6);
        System.out.println(obiectUnu);
        System.out.println(obiectDoi);
    }
}

```

Primul constructor al clasei C2 apelează cel de-al doilea constructor adăugând ca argument atributul y. Al doilea constructor al clasei C2 apelează constructorul clasei C1, inițializând atributul x. Dacă primul constructor al clasei C2 ar fi avut un apel explicit de forma `this(x, z)`, atunci am fi obținut eroare la compilare, deoarece în apelul explicit al unui constructor nu putem avea variabile instanță. Programul va afișa la execuție:

```

x = 4, y = 2, z = 2
x = 5, y = 2, z = 6

```

Supraîncărcarea constructorilor este identică cu cea a metodelor. Supraîncărcarea este rezolvată în timpul compilării pentru fiecare expresie de creare a instanței clasei.

Dacă o clasă nu conține nici o declarație de constructor, atunci se creează de către compilator un *constructor implicit* fără parametri, astfel:

- dacă este vorba de clasa `Object` (superclasa tuturor claselor), atunci constructorul implicit are corpul vid;
- altfel, constructorul implicit nu are parametri și apelează constructorul super-clasei fără argumente.

Se obține eroare la compilare în cazul în care compilatorul creează un constructor implicit și superclasa acesteia are constructori, însă nici unul fără parametri.

Exemplul 3.2.47. Următoarele clase Java scot în evidență apelul constructorului implicit pentru clasa C2.

```

class C1 {
    int x = 1;
    C1(int x) { this.x = x; }
}

```

```

class C2 extends C1 {
    int y = 2;
}

```

Acestea vor conduce la eroare de compilare, deoarece constructorul implicit al clasei C2 va apela constructorul implicit al clasei C1 care nu există, deoarece a fost definit doar un singur constructor cu parametru.

Dacă o clasă este declarată `public`, atunci constructorul implicit are acces `public`; altfel, în cazul lipsei modifierului de acces, constructorul implicit are acces implicit (eng. *friendly*).

Exemplul 3.2.48. Următoarea declarație de clasă Java:

```

public class C {
    int x;
}

```

este echivalentă cu declarația:

```

public class C {
    int x;
    public C() { super(); }
}

```

Continuăm cu prezentarea relației dintre constructorii clasei derivate și ai superclasei acesteia. Când se creează un obiect al clasei derivate, acesta conține un sub-obiect al clasei de bază. Acest sub-obiect al clasei de bază trebuie inițializat corect prin apelul explicit al constructorului clasei de bază care poate inițializa sub-obiectul clasei de bază. În cazul constructorilor implicați (fără parametri), Java inserează apelul constructorului implicit al clasei de bază înaintea apelului efectiv al constructorului clasei curente. În cazul constructorilor cu parametri, trebuie făcut apelul constructorului clasei de bază în mod explicit (utilizând cuvântul rezervat `super`), altfel se obține eroare la compilare.

Exemplul 3.2.49. Următorul program Java evidențiază ordinea apelurilor constructorilor implicați pentru clasele derivate.

```

class C1 {
    C1() { System.out.println("Constructorul clasei C1"); }
}
class C2 extends C1 {
    C2() { System.out.println("Constructorul clasei C2"); }
}
public class OrdineConstructori extends C2 {
    OrdineConstructori() {
        System.out.println(
            "Constructorul clasei OrdineConstructori");
    }
}

```



```

    }
    public static void main(String[] args) {
        OrdineConstructori x = new OrdineConstructori();
    }
}

```

Clasa `OrdineConstructori` este derivată din `C2`, care la rândul ei este derivată din `C1`. Astfel, se va apela întâi constructorul superclasei `C1`, apoi `C2` și în final al clasei `OrdineConstructori`. Așadar, programul va afișa la execuție:

```

Constructorul clasei C1
Constructorul clasei C2
Constructorul clasei OrdineConstructori

```

Exemplul 3.2.50. Următorul program Java scoate în evidență apelurile constructorilor cu parametri pentru clasele derivate. Spre deosebire de Exemplul 3.2.46, clasa derivată va trebui să conțină ca primă instrucțiune în constructorul său un apel al constructorului explicit al clasei derivate folosind `super`, în caz contrar se obține eroare la compilare.

```

class C1 {
    int x = 1;
    C1 (int x) { this.x = x; }
}
class C2 extends C1 {
    int x = 2, y = 3;
    C2(int x) { super(x); }
    C2(int x, int y) { super(x); this.y = y; }
}
public class ApelExplicitConstructor {
    public static void main(String args[]) {
        C1 obiectUnu = new C1(4);
        C2 obiectDoi = new C2(5);
        C2 obiectTrei = new C2(6, 7);
        System.out.print(obiectUnu.x + " ");
        System.out.print(obiectDoi.x + " " + obiectDoi.y + " ");
        System.out.print(obiectTrei.x + " " + obiectTrei.y);
    }
}

```

Dacă în declarațiile constructorilor clasei `C2` ar fi lipsit instrucțiunea `super(x)`, atunci s-ar fi obținut eroare la compilare. La crearea unui obiect din `C2`, întâi se apelează constructorul din clasa `C1`, apoi cel din clasa `C2`. Rezultă imediat că pentru toate obiectele din `C2`, atributul `x` va fi întotdeauna egal cu 2 (la apelul `super(x)` este asignat întâi cu 1, în cele din urmă cu 2). Astfel, programul va afișa la execuție:

```

4 2 3 2 7

```

3.2.9. Clase interioare

Clasele interioare (eng. *inner classes*) sau *imbricate* au fost introduse începând cu versiunea JDK 1.1. Uneori clasele interioare pot da programelor mai multă claritate. O clasă interioară este o clasă definită în interiorul altei clase. Accesul la membrii clasei interioare se face analog cu cei de la clasele obișnuite. De exemplu, dacă o clasă interioară este calificată `private`, atunci putem să o folosim doar în clasa exterioară unde este definită. Clasele interioare sunt importante pentru că permit gruparea claselor care sunt logic asemănătoare fără a coincide cu compunerea claselor. Clasele interioare sunt folosite frecvent în construirea obiectelor `listener` pentru rezolvarea evenimentelor.

Exemplul 3.2.51. Următorul program Java pune în evidență accesul la o dată `private` dintr-o clasă interioară:

```

class Exterior {
    private int x = 2;
    public class Interior {
        private int y = 5;
        public void metodaInterioara() {
            System.out.println("x = " + x);
            x = 3;
            System.out.println("x = " + x);
            System.out.println("y = " + y);
        }
    } // sfarsitul clasei Interior
    public void metodaExterioara() {
        System.out.println("x = " + x);
    }
}

```

La compilarea acestui program, se vor forma două clase: `Exterior` și `Exterior$Interior`. Codurile binare Java (byte code) corespunzătoare generate pe disc vor fi `Exterior.class` și `Exterior$Interior.class` (se utilizează caracterul „\$” în loc de „.” pentru a nu fi confundat cuvântul `Interior` cu o extensie a unui program). Când este creată o instanță a clasei interioare, trebuie să existe de obicei o instanță a clasei exterioare. Această instanță a clasei exterioare este accesibilă din obiectul interior. Astfel, în exemplul precedent, variabilele `x` și `y` sunt accesibile din clasa `Interior`. Programul va afișa la execuția metodei `metodaInterioara()`:

```

x = 2
x = 3
y = 5

```

Accesibilitatea membrilor clasei exterioare este posibilă, deoarece clasa interioară are de fapt o referință ascunsă către instanța clasei exterioare care a fost contextul

curent când clasa interioară a fost creată. De fapt, aceasta asigură că și clasa interioară, și cea exterioară aparțin împreună, și nu faptul că instanța interioară este doar un membru al instanței exterioare.

Exemplul 3.2.52. Uneori dorim să creăm o instanță a unei clase interioare dintr-o metodă statică sau când nu avem la dispoziție un obiect `this` disponibil. Iată cum putem defini un obiect dintr-o clasă interioară fără să utilizăm cuvântul rezervat `this`:

```
public static void main(String args[]) {
    Exterior.Interior obiect = new Exterior().new Interior();
    obiect.metodaInterioara();
}
```

Acesta poate fi „decompactat” în mod echivalent:

```
public static void main(String args[]) {
    Exterior obiectTemporar = new Exterior();
    Exterior.Interior obiect = obiectTemporar.new Interior();
    obiect.metodaInterioara();
}
```

Putem defini și *clase interioare statice*. Acestea nu mai au referința către clasa exterioară. Deci metodele unei clase interioare statice nu pot accesa variabile instanță ale clasei exterioare.

Putem defini, de asemenea, clase în interiorul unei metode din altă clasă (eng. *local nested class*). Putem crea obiecte ale unei clase interioare locale doar în metoda în care apare definiția clasei. Aceasta este utilă atunci când calculul într-o metodă necesită folosirea unei clase specializate care nu este cerută sau folosită în altă parte. În plus față de clasele definite în interiorul claselor, avem două aspecte particulare:

1. Un obiect creat într-o clasă interioară unei metode are „alt” acces la variabilele metodei exterioare;
2. Este posibil să creăm o clasă anonimă (adică o clasă fără nume specificat).

Prezentăm pe rând cele două aspecte particulare prezentate mai sus:

1. Orice variabilă, locală sau parametru formal, poate fi accesată de clasa interioară numai dacă este calificată `final` (adică este o constantă). Permițând accesul numai la variabilele `final`, este posibil să copiem valorile acestor variabile chiar în obiect, extinzând astfel viața obiectelor clasei interioare.

Exemplul 3.2.53. Următorul program Java pune în evidență apelul unei metode a unei clase interne altei clase.

```
public class ClasaInterioara {
    public static void main(String args[]) {
        Exterior obiectExterior = new Exterior();
        // apelam o metoda a unei clase interne unei alte clase
```

```
        Exterior.Interior obiectInterior = obiectExterior.new
        Interior();
        obiectInterior.metodaInterioara();
        // apelam o metoda a unei clase care construiește un obiect
        // al altei clase
        obiectExterior.f(4, 3);
    }
}

class Exterior {
    private int x = 2;
    public class Interior {
        private int y = 5;
        public void metodaInterioara() {
            System.out.println("x = " + x);
            System.out.println("y = " + y);
        }
    }
    public void metodaExterioara() {
        System.out.println("x = " + x);
    }
    public void f(int xx, final int yy) {
        int a = xx + yy;
        final int b = xx - yy;
        class Interioara {
            public void metodaInterioara() {
                System.out.println("x = " + x);
                System.out.println("b = " + b);
            }
        }
        Interioara obiectInteriorSecund = new Interioara();
        obiectInteriorSecund.metodaInterioara();
    }
}
```

`obiectInterior` este un obiect al clasei `Exterior.Interior` și deci apelul lui `metodaInterioara()` va implica afișarea atributelor private `x` și `y` (accesibile clasei interioare). Apelarea metodei `f()` asupra lui `obiectExterior` va implica crearea lui `obiectInteriorSecund`, căruia i se aplică `metodaInterioara()`. Aceasta va afișa valorile atributului `final b` și ale atributului `x` al clasei `Exterior`. Nu putem accesa variabila locală `a` a metodei `f()` dintr-o clasă interioară, în caz contrar se obține eroare la compilare. Astfel, programul va afișa la execuție:

```
x = 2
y = 5
```

```
x = 2
b = 1
```

2. Pentru clase anonime, nu putem utiliza operatorul `new` în modul bine cunoscut. De fapt, clasele anonime sunt definite în locul unde sunt construite.

Exemplul 3.2.54. Programul următor reprezintă un applet în care utilizăm clase anonime:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Buton extends Applet {
    private Button buton;
    public void init() {
        buton = new Button("Apasa-ma");
        add(buton);
        buton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("A aparut o actiune");
            }
        });
    } // sfarsitul definitiei metodei init()
} // sfarsitul definitiei clasei Buton
```

Apelul lui `new` este urmat de definiția clasei (anonime). Clasa nu are nume, dar este referită printr-un nume de interfață (fără să folosească cuvântul rezervat `implements`). Nu putem instanția clase anonime decât în modul de mai sus. Este recomandat ca și clasele anonime să fie mici. O clasă anonimă poate fi o subclasă a unei clase explicite sau poate implementa o singură interfață explicită. Dacă o clasă anonimă extinde o clasă existentă, atunci argumentele constructorului superclasei trebuie puse în partea de argumente a expresiei `new`. De exemplu, putem avea în programul de mai sus expresia:

```
new Button("Apasa") { // modificari pentru Button ... }
```

Execuția appletului prezentat conduce la afișarea:



3.2.10. Distrugerea obiectelor și eliberarea memoriei

Obiectele (instanțele unei clase) se creează cu operatorul `new` sau cu metoda `newInstance()` a clasei `Class`. Marea majoritate a programelor Java în care se instanțiază clase folosesc operatorul `new`. Exemplul 2.4.8 pune în evidență instanțierea unei clase cu metoda `newInstance()`. În momentul creării unei instanțe pentru o clasă, se alocă memorie pentru obiectul nou creat. Acesta va avea o adresă în memorie. Dacă ulterior obiectului creat i se aplică din nou operatorul `new` sau cu metoda `newInstance()`, atunci se alocă o nouă zonă de memorie care va avea bineînțeles altă adresă. Spațiul vechi de memorie se eliberează automat. Acest mecanism se numește „colectarea gunoaielor” (eng. *garbage collection*), numit așa în ideea reutilizării spațiului de memorie.

Exemplul 3.2.55. Programul de mai jos pune în evidență realocarea memoriei pentru un obiect folosind operatorul `new` și metoda `newInstance()` din clasa `Class`.

```
class C1 {
    int x = 10;
    C1() { System.out.println("Constructor fara argumente.");
    }
    C1(int x) {
        this.x = x;
        System.out.println("Constructor cu un argument.");
    }
}

public class TestDealocare {
    public static void main(String args[]) {
        C1 obiectUnu = new C1(8);
        System.out.println("adresa lui obiectUnu = " +
            obiectUnu.hashCode());
        obiectUnu = new C1(6);
        System.out.println("adresa lui obiectUnu = " +
            obiectUnu.hashCode());
        Class oClasa = null;
        try {
            oClasa = Class.forName("C1");
        }
        catch (ClassNotFoundException e) {
            System.out.println("Exceptie: n-am gasit clasa C1. " + e);
        }
        try {
            obiectUnu = (C1) oClasa.newInstance();
        }
    }
}
```

```

catch (InstantiationException e) {
    System.out.println(
        "Exceptie: nu putem instantia clasa C1." + e);
}
catch (IllegalAccessException e) {
    System.out.println(
        "Exceptie: acces ilegal la clasa C1." + e);
}
System.out.println("adresa lui obiectUnu = " +
    obiectUnu.hashCode());
}
}

```

Observăm că primul obiect (obiectUnu) al clasei C1 a avut o „viață” foarte scurtă. Mai precis, s-a realocat alt spațiu de memorie, iar vechiul spațiu s-a eliberat, eventual spre o reutilizare ulterioară. Al doilea obiect (numit tot obiectUnu) are repartizată altă zonă de memorie. Din nou, se realocă spațiu pentru al treilea obiect numit la fel, urmând ca vechiul spațiu să fie eliberat de către colectorul de gunoaie (eng. *garbage collector*). Astfel, programul va afișa la execuție:

```

Constructor cu un argument.
adresa lui obiectUnu = 6129586
Constructor cu un argument.
adresa lui obiectUnu = 7852340
Constructor fara argumente.
adresa lui obiectUnu = 4711763

```

Pentru a distruge un obiect, urmând ca spațiul de memorie să fie eliberat de *garbage collector*, îl asignăm cu `null`, desemnând un obiect care nu există.

Exemplul 3.2.56. Considerăm următorul cod Java (presupunem că am definit o clasă C1 care are o metodă publică `afiseaza()`):

```

C1 obiect;
obiect.afiseaza();

```

Programul se va opri la execuție cu un mesaj de eroare de tip „*null pointer exception*”. Aceasta deoarece am declarat un obiect, dar nu l-am creat folosind `new` sau `newInstance()`. Obiectul nu există, el are valoarea `null`. Dacă dorim să distrugem explicit un obiect (deși un obiect nefolosit este automat distrus de sistemul *garbage collector*), putem să îi asignăm valoarea `null` acelui obiect. Acesta va fi distrus la următorul apel al „colectorului de gunoaie automat”.

Există însă cazuri când alocarea zonei de memorie nu s-a făcut cu `new` sau `newInstance()`, de exemplu metode (specificate native) scrise în alte limbaje de programare (cum ar fi C/C++) care conțin apeluri ale funcțiilor `malloc()`, `calloc()` etc. De asemenea, există cazuri în care se solicită eliberarea unor alte resurse decât cele de memorie, de exemplu, a unor conexiuni de fișiere sau a unor

conexiuni de rețea. Colectorul de gunoaie poate doar să elibereze automat memoria alocată cu operatorul `new` sau cu metoda `newInstance()`. Pentru a rezolva și aceste cazuri speciale, Java pune la dispoziție metoda:

```
protected void finalize() throws Throwable
```

moștenită din clasa `Object`. Vom defini această metodă pentru clasa cu care lucrăm. Când *garbage collector*-ul este gata să elibereze spațiul de memorie folosit de obiect, acesta va apela mai întâi `finalize()` și abia după aceea se va elibera memoria. Problema care se pune este că nu se știe când obiectul va fi „curățat”, deci nici când se va face apelul metodei `finalize()`. Se știe doar că aceasta va avea loc înainte ca zona de memorie respectivă să fie reutilizată. Metoda `finalize()` nu poate fi apelată direct de programator.

În clasa predefinită `System` există metodele statice:

```

public static void gc();
public static void runFinalization();
public static void runFinalizersOnExit(boolean valoare);

```

Metoda `gc()` forțează execuția *garbage collector*-ului și se utilizează împreună cu `runFinalization()`, care implică execuția metodelor `finalize()` a obiectelor ce nu mai sunt referite de nici o referință, dar care nu au fost încă finalizate. Metoda `runFinalizersOnExit()` setează indicatorul (eng. *flag*) care indică mașinii virtuale Java dacă se execută sau nu metodele `finalize()` ale obiectelor încă neeliberate la oprirea mașinii virtuale. Metoda `runFinalizersOnExit()` este învechită fiind inerent nesigură, deoarece, apelată pentru obiecte accesate de fire de execuție, poate rezulta o comportare haotică sau chiar blocare (eng. *deadlock*).

Exemplul 3.2.57. Programul de mai jos pune în evidență crearea unui număr mare de obiecte ale unei clase și apoi eliberarea memoriei asociate acestora:

```

class C1 {
    static boolean pornitColectorul = false;
    static boolean gata = false;
    static int numarInstante = 0;
    static int finalizate = 0;
    final int NR_INST_MAX = 32;
    int i;
    C1() {
        i = ++ numarInstante;
        if (numarInstante == NR_INST_MAX)
            System.out.println(
                "A fost creat numarul maxim de instante.");
    }
    protected void finalize() {
        if (!pornitColectorul) {
            pornitColectorul = true;

```

```

        System.out.println("Incepem sa finalizam dupa " +
            numarInstante + " instante create.");
    }
    if (i == NR_INST_MAX) {
        System.out.println("Finalizam ultimul obiect si " +
            "oprim crearea obiectelor de tip C1.");
        gata = true;
    }
    finalize++;
    if (finalize >= numarInstante)
        System.out.println("Toate cele " + finalize +
            " obiecte au fost finalizate.");
}

}

public class GarbageCollection {
    public static void main(String[] args) {
        while (!C1.gata) {
            new C1();
        }
        System.out.println("S-au creat in total " +
            C1.numarInstante + " instante; total finalizate = " +
            C1.finalize);
        // urmeaza dealocarea memoriei
        System.out.println("Apelam gc():");
        System.gc();
        System.out.println("Apelam runFinalization():");
        System.runFinalization();
        System.out.println("Am terminat.");
    }
}

```

Clasa C1 suprascrie metoda finalize() din clasa Object. Se vor crea un număr mare de obiecte până când variabila booleană gata va deveni true, lucru care se întâmplă doar când colectorul de gunoarie va apela automat metoda finalize(). După ce variabila gata devine true, se oprește iterația de creare a noilor obiecte din clasa C1 și se va afișa numărul total de obiecte create, respectiv finalizate. Astfel, o execuție posibilă a programului de mai sus este (în funcție de „viteza de reacție” a colectorului de gunoarie):

```

A fost creat numarul maxim de instante.
Incepem sa finalizam dupa 4222 instante create.
Finalizam ultimul obiect si oprim crearea obiectelor de tip C1.
S-au creat in total 13742 instante; total finalizate = 4222
Apelam gc():

```

```

Toate cele 13742 obiecte au fost finalizate.
Apelam runFinalization():
Am terminat.

```

3.2.11. Împărțirea unei aplicații în fișiere

Dacă un program constă doar dintr-o singură clasă, codul sursă Java este plasat într-un fișier cu același nume ca al clasei și cu extensia .java. Orice instrucțiune de tip import trebuie plasată înaintea declarației clasei. Compilatorul transformă codul sursă Java în cod binar Java, plasat într-un fișier cu numele clasei, dar cu extensia .class. Dacă programul constă în mai mult de o clasă există două alternative:

1. Plasarea tuturor claselor într-un singur fișier.
2. Plasarea fiecărei clase într-un fișier separat (cu același nume cu numele clasei).

Dacă toate clasele sunt plasate într-un singur fișier, prima clasă din fișier trebuie declarată public, iar celelalte nu. Numele fișierului trebuie să corespundă cu prima clasă a fișierului. În cele ce urmează, vom oferi o soluție de a împărți o clasă (eventual prea mare) în două clase de dimensiuni mai reduse. Presupunem că inițial avem:

```

class A {
    ...
    public static void main(String args[]) {
        ...
        apel g();
        ...
    }
    void f() {
        ...
    }
    void g() {
        ...
    }
}

```

Dorim ca metoda g() să fie în altă clasă, însă ea are acces la datele private și public ale clasei A. O posibilă împărțire echivalentă în două clase este:

```

class A {
    ...
    B obiectDinB;
    public static void main(String args[]) {
        ...
        obiectDinB() = new B(this);
        obiectDinB.g();
    }
}

```

```

void f() {
    ...
}
}
class B {
    A obiectDinA;
    ...
    public B(A obiectDinA) {
        this.obiectDinA = obiectDinA;
    }
    void g() {
        //accesul la membrii public din A se face direct
        obiectDinA.membruPublicDinA;
        //accesul la datele private din clasa A se face folosind
        //functii de acces public
        obiectDinA.metodaPublica();
    }
}

```

Exemplul 3.2.58. Programul de mai jos pune în evidență o împărțire a unei clase în două clase mai mici echivalente:

```

public class C1 {
    public static void main(String args[]) {
        A obA = new A(1);
        B obB = new B(2, obA);
        obA = new A(1, obB);
        try {
            obB.get().afiseaza();
            obA.get().afiseaza();
        }
        catch (NullPointerException e) {
            System.out.println("Eroare: " + e.getMessage());
        }
    }
}

class A {
    private int x;
    private B obiectDinB;
    A(int x) {
        this.x = x;
        this.obiectDinB = null;
    }
}

```

```

A(int x, B obiectDinB) {
    this.x = x;
    this.obiectDinB = obiectDinB;
}

B get() throws NullPointerException {
    if (obiectDinB == null)
        throw new NullPointerException(
            "Obiectul din B este null\n");
    else
        return obiectDinB;
}

void afiseaza() {
    System.out.println("x = " + x);
}

}

class B {
    private int y;
    private A obiectDinA;

    B(int y) {
        this.y = y;
        this.obiectDinA = null;
    }

    B(int y, A obiectDinA) {
        this.y = y;
        this.obiectDinA = obiectDinA;
    }

    A get() throws NullPointerException {
        if (obiectDinA == null)
            throw new NullPointerException(
                "Obiectul din A este null\n");
        else
            return obiectDinA;
    }

    void afiseaza() {
        System.out.println("y = " + y);
    }
}

```

Clasele A, respectiv B au ca atribute *x* și *obiectDinB*, respectiv *y* și *obiectDinA*. Cei doi constructori ai acestor clase au drept scop inițializarea acestor atribute și crearea legăturii duble între cele două instanțe ale claselor A și B. În plus, clasele A și B au câte o metodă de acces care returnează referința către obiectul celeilalte clase și

o metodă de tipărire a atributelor *x*, respectiv *y*. Pentru testarea acestui exemplu, clasa C1 creează trei obiecte, două din clasa A (din care unul cu „durată de viață” foarte scurtă) și unul din clasa B. Astfel, programul va afișa la execuție:

```
x = 1
y = 2
```

3.3. Interfețe

O interfață grupează mai multe metode și date membre publice. Sunt definite doar prototipurile metodelor, implementările urmând a fi scrise în clasa care va implementa respectiva interfață. Gruparea metodelor publice este utilă pentru a da aceeași modalitate de lucru cu anumite clase. Există și interfețe care nu posedă metode sau date membre, acestea având rol de indicator (eng. *flag*).

O clasă poate implementa mai multe interfețe, dar poate moșteni doar o singură clasă. Relația de moștenire este valabilă și pentru interfețe. Prin implementarea mai multor interfețe se simulează moștenirea multiplă (întâlnită în limbajul C++).

O declarație de interfață introduce un tip referință ai cărui membri sunt constante și metode abstracte. Acest tip nu posedă implementări (metodele nu sunt definite), dar anumite clase pot implementa aceste interfețe furnizând definiții pentru metodele abstracte. Sintaxa generală a declarației unei interfețe este:

```
[public | abstract] interface <Identificator> [extends
<ListaInterfete>] {
    [<DeclaratiiMembriInterfata>]
}
```

unde:

- <Identificator> reprezintă numele interfeței;
- <ListaInterfete> are forma generală: <Interfata1>, ..., <InterfataN>, unde <InterfataI> reprezintă identificatori (nume) de interfețe, pentru I de la 1 la N;
- <DeclaratiiMembriInterfata> are forma generală: <DeclaratieMembru1> ... <DeclaratieMembruK>, unde <DeclaratieMembruI>, pentru I de la 1 la K, poate fi: <DeclaratieConstanta> sau <DeclaratieMetodaAbstracta>

Numele unei interfețe (<Identificator>) nu poate coincide cu numele altei clase sau interfețe din același pachet, altfel apare eroare la compilare. Ca și în cazul claselor, domeniul numelui unei interfețe este întreg pachetul în care este declarat. Fiecare interfață este implicit abstractă, deci este considerat învechit modificatorul *abstract* în declarația unei interfețe.

Dacă declarația unei interfețe conține clauza *extends*, atunci interfața va moșteni toate metodele și constantele interfețelor enumerate în <ListaInterfete>, care se

numesc *superinterfețele directe*. Orice clasă care implementează interfața declarată va trebui să implementeze toate superinterfețele accesibile clasei.

Superinterfețele din clauza *extends* trebuie să fie accesibile, altfel apare eroare la compilare. Nu există o superinterfață generală (pentru toate interfețele), cum este clasa *Object* pentru superclase.

Ca și în cazul claselor, relația de *derivare a interfețelor* (indirectă) este închiderea tranzitivă a relației de derivare directă (definiția este dată în secțiunea 3.2.4).

Membrii unei interfețe sunt acei membri moșteniți din superinterfețele directe și acei membri declarați în interfață. Interfața moștenește toți membrii superinterfețelor, cu excepția atributelor ascunse și a metodelor suprascrise. Domeniul numelui unui membru declarat într-un tip interfață este corpul declarației interfeței. Toți membrii unei interfețe sunt implicit public. Astfel, dacă membrii sunt declarați public, atunci aceștia sunt accesibili și în afara pachetului.

3.3.1. Declarațiile atributelor unei interfețe

Atributele unei interfețe sunt de fapt constante și au sintaxa generală:

```
[public | static | final] <Tip> <DeclaratieVariabilal> [, ...,
<DeclaratieVariabilaN>];
```

unde:

- modificatorii *public*, *static* și *final* sunt implicați și specificarea lor este redundantă;
- <DeclaratieVariabilal>, cu I = 1, ..., N, poate fi:
 - <Identificator> = <InitalizatorVariabila> sau
 - <Identificator> [] [[]...[]] = <InitalizatorTablou>

Spre deosebire de clasă, declarația unui atribut într-o interfață nu poate conține modificatorii *synchronized*, *transient* sau *volatile*.

Fiecare atribut din corpul unei interfețe trebuie să aibă o expresie de inițializare, care nu este obligatoriu să fie constantă. Inițializatorul variabilei este evaluat și asignarea are loc o singură dată, când se inițializează interfața. O expresie de inițializare nu poate conține o referință prin nume simplu la același atribut sau la alt atribut a cărui declarație apare mai târziu în aceeași interfață. La fel, dacă expresia de inițializare conține cuvintele rezervate *this* sau *super*, atunci apare o eroare la compilare.

Este posibil pentru o interfață să moștenească mai multe atribute cu același nume de la mai multe interfețe, însă încercarea de a ne referi cu nume simplu la un astfel de atribut conduce la eroare de compilare deoarece referința este ambiguă.

Exemplul 3.3.1. Următorul program Java scoate în evidență situația când un atribut apare cu același nume în două interfețe distincte:

```
interface I1 {
    int j = 3;
```

```

    int [] i = {j + 1, j + 2};
}
interface I2 {
    int j = 5;
    int [] i = {j + 2, j + 4};
}
public class TestAtribute implements I1, I2 {
    public static void main(String[] args) {
        System.out.println(I1.j);
        System.out.println(I2.i[1]);
        System.out.println(I1.i[0]);
    }
}

```

Dacă s-ar fi încercat accesarea atributelor *i* și *j* din interfețele *I1* și *I2* fără numele calificat al interfețelor, s-ar fi obținut eroare la compilare. Programul este corect și va afișa:

```

3
9
4

```

Dacă două atribute cu același nume sunt moștenite de o interfață, atunci se obține *ambiguitatea atributului*. Această situație poate apărea, de exemplu, când două super-interfețe directe declară acest atribut. Încercarea de accesare a acestui atribut conduce la eroare de compilare. Evident, este posibilă accesarea folosind nume calificat (numele interfeței dorite).

Invers, o altă situație este când un singur atribut este moștenit de mai multe din aceeași interfață (se mai numește *moștenire multiplă*). Această situație se poate obține, de exemplu, când interfața *I0* este superinterfață pentru alte două noi interfețe *I1* și *I2*, care la rândul lor sunt superinterfețe pentru o altă interfață *I3*. Astfel, atributele din interfața *I0* sunt moștenite de mai multe ori, însă se păstrează doar o singură copie, deci accesarea acestora nu determină eroare la compilare.

Exemplul 3.3.2. Următorul program Java scoate în evidență ambiguitatea atributelor și moștenirea multiplă a atributelor:

```

interface I0 {
    int x = 3;
}
interface I1 extends I0 {
    int [] y = {x + 1, x + 2};
}
interface I2 extends I0 {
    int [] y = {x + 3, x + 4};
}

```

```

interface I3 extends I1, I2 {
    /* se face referire la primul element
       din tabloul y din interfata I1 */
    int z = I1.y[0];
    /* se face referire la al doilea element
       din tabloul y din interfata I2 */
    int t = I2.y[1];
}
public class TestAtributeAmbigue implements I3 {
    public static void main(String[] args) {
        System.out.println(x);
        System.out.println(z);
        System.out.println(t);
    }
}

```

Atributul *y* este prezent atât în interfața *I1*, cât și în interfața *I2*. Accesarea simplă a acestuia în interfața *I3* și în clasa *TestAtributeAmbigue* va conduce la eroare de compilare. Pentru a evita eroarea de compilare, trebuie utilizat numele interfeței din care face parte atributul (*I1* sau *I2*). Atributul *x* este moștenit de două ori în interfața *I3*, deoarece *x* este prezent ca atribut în interfața *I1* și *I2*. Astfel, programul va afișa la execuție:

```

3
4
7

```

3.3.2. Declarațiile metodelor unei interfețe

Metodele unei interfețe sunt asemănătoare cu metodele unei clase, cu excepția faptului că structura *<CorpMetodă>* este ; și că nu permit apariția tuturor modificatorilor de clasă. Spre deosebire de metodele unei clase, declarația unei metode într-o interfață nu poate conține modificatorii *static*, *final*, *synchronized* sau *native*. Sintaxa generală a declarației metodei unei interfețe este:

```

[public | abstract] <TipRezultat> <Identificator>
([<ListaParametriFormali>])
throws <TipClasă1> [, ..., <TipClasăN>];

```

unde:

- modificatorii *public* și *abstract* sunt implicați și specificarea lor este redundantă;
- *<Identificator>* reprezintă numele metodei.

Declarația unei metode *suprascrive* toate metodele cu aceeași semnătură din super-interfețele sale care sunt accesibile codului interfeței. În acest caz, acestea trebuie să

întoarce același tip, și dacă au clauze throws, nu trebuie să intre în conflict. Astfel, o interfață moștenește toate metodele ne-suprascrise ale superinterfețelor sale. Metodele declarate în interfețe sunt abstracte și deci nu au implementare. Cum metodele suprascrise au aceeași semnătură (și compilatorul le forțează să returneze același tip), înseamnă că singurul loc unde pot să difere este în lista de excepții care pot fi aruncate de metoda suprascrisă (eventual restrângând lista acestora).

Exemplul 3.3.3. Următorul program Java evidențiază suprascrierea metodelor unei interfețe relativ la lista de excepții care pot fi aruncate:

```
class ExceptiaNoastraUnu extends Exception {
    ExceptiaNoastraUnu() { super(); }
}
class ExceptiaNoastraDoi extends Exception {
    ExceptiaNoastraDoi() { super(); }
}
interface I1 {
    int f(int x) throws ExceptiaNoastraUnu, ExceptiaNoastraDoi;
}
interface I2 extends I1 {
    int f(int x) throws ExceptiaNoastraDoi;
}
class C1 implements I1 {
    public int f(int x) throws ExceptiaNoastraUnu,
        ExceptiaNoastraDoi {
        if (x < 0) throw new ExceptiaNoastraUnu();
        else throw new ExceptiaNoastraDoi();
    }
}
class C2 implements I2 {
    public int f(int x) throws ExceptiaNoastraDoi {
        if (x < 0) throw new ExceptiaNoastraDoi();
        return x;
    }
}
public class TestSuprascriereMetode {
    public static void main(String[] args) {
        C1 obiectUnu = new C1();
        C2 obiectDoi = new C2();
        try {
            System.out.println(obiectDoi.f(2));
            System.out.println(obiectUnu.f(-2));
            System.out.println(obiectDoi.f(-2));
            System.out.println(obiectUnu.f(2));
        }
    }
}
```

```
catch (ExceptiaNoastraUnu e) {
    System.out.println("A aparut o exceptie: " + e);
}
catch (ExceptiaNoastraDoi e) {
    System.out.println("A aparut o exceptie: " + e);
}
}
```

Metoda `f()` din clasa `C2` implementează metoda abstractă `f()` din interfața `I2`. Încercarea de a implementa în clasa `C2` metoda `f()` din interfața `I1` conduce la o eroare de compilare. Deși clasa `TestSuprascriereMetode` presupune patru afișări de texte, se aruncă o excepție încă de la a doua afișare la consola de ieșire. Astfel, programul va afișa la execuție:

```
2
A aparut o exceptie: ExceptiaNoastraUnu
```

Dacă două metode ale unei interfețe (declarate în aceeași interfață sau moștenite) au același nume, dar diferite semnături, atunci spunem că numele metodei este *supraîncărcat*.

Exemplul 3.3.4. Următorul program Java scoate în evidență supraîncărcarea metodelor unei interfețe:

```
interface I1 {
    int f(int x);
}
interface I2 extends I1 {
    float f(float x);
    double f(double x);
}
class C1 implements I2 {
    public int f(int x) { return x; }
    public float f(float x) { return x; }
    public double f(double x) { return x; }
}
public class TestSupraincarcareMetode {
    public static void main(String[] args) {
        C1 obiect = new C1();
        System.out.println(obiect.f(2));
        System.out.println(obiect.f(4.5));
        System.out.println(obiect.f(2.5f));
    }
}
```

Numele metodei `f()` este supraîncărcat cu trei semnături diferite în interfața `I2`, clasa `C1` trebuind să implementeze toate cele trei metode. Astfel, programul va afișa:

```
2
4.5
2.5
```

3.3.3. Moștenire multiplă prin intermediul interfețelor

Am văzut că Java nu permite moștenirea multiplă a claselor. Structura claselor este un arbore (și nu un graf de tip rețea) cu rădăcină, în care o clasă poate avea oricâte subclase, dar numai o superclasă. Există, totuși, un mod de simulare a facilităților moștenirii multiple în Java utilizând interfețe. O clasă poate implementa orice număr de clase (interfețe). Această secțiune prezintă o aplicație ceva mai mare decât cele de până acum.

Exemplul 3.3.5. Programul Java de mai jos prezintă o aplicație în care se folosesc interfețe. Avem o interfață `DispozitiveElectronice`, o clasă de bază `Locuinta`, două clase derivate (`TV` și `Aspirator`) din clasa de bază. Fișierul de aplicații este intitulat `AplicatieElectronice.java`.

```
// Fisierul DispozitiveElectronice.java
interface DispozitiveElectronice {
    public void setTimpFolosinta(int timp);
    public int getTimpFolosinta();
    public int getPutere();
}
```

```
// Fisierul Locuinta.java
abstract class Locuinta {
    protected int volum = 1;
    private boolean on;

    public void porneste() {
        on = true;
    }

    public void opreste() {
        on = false;
    }

    public abstract void schimbaVolum(int diferenta);
}
```

```
// Fisierul TV.java
```

```
class TV extends Locuinta implements DispozitiveElectronice {
    private int timp = 0;

    public void schimbaVolum(int diferenta) {
        volum = volum + diferenta;
    }

    public void setTimpFolosinta(int timp) {
        this.timp = timp;
    }

    public int getTimpFolosinta() {
        return timp;
    }

    public int getPutere() {
        return 100;
    }

    public int getVolum() {
        return volum;
    }
}
```

```
// Fisierul Aspirator.java
class Aspirator extends Locuinta implements
    DispozitiveElectronice {
    private int timp = 0;

    public void schimbaVolum(int diferenta) {
        volum = volum + diferenta;
    }

    public void setTimpFolosinta(int timp) {
        this.timp = timp;
    }

    public int getTimpFolosinta() {
        return timp;
    }

    public int getPutere() {
        return 500;
    }
}
```

```

    public int getVolum() {
        return volum;
    }
}

// Fisierul AplicatieElectronice.java
import java.io.*;

public class AplicatieElectronice {
    private BufferedReader tastatura;
    private int totalPutere;
    private int timp, volum;

    public static void main(String[] args) {
        AplicatieElectronice obiect = new AplicatieElectronice();
        TV tv = new TV();
        obiect.citesteTimp();
        tv.setTimpFolosinta(obiect.timp);
        tv.schimbaVolum(1);
        obiect.calculeazaPutereConsumata(tv);
        obiect.afiseaza("tv", tv.getVolum(),
            tv.getTimpFolosinta(), obiect.totalPutere);
        tv.setTimpFolosinta(0);
        Aspirator aspirator = new Aspirator();
        obiect.citesteTimp();
        aspirator.setTimpFolosinta(obiect.timp);
        aspirator.schimbaVolum(2);
        obiect.calculeazaPutereConsumata(aspirator);
        obiect.afiseaza("aspirator", aspirator.getVolum(),
            aspirator.getTimpFolosinta(), obiect.totalPutere);
        aspirator.setTimpFolosinta(0);
    }

    private void citesteTimp() {
        try {
            tastatura = new BufferedReader(
                new InputStreamReader(System.in), 1);
            System.out.print("Dati timpul (in minute): ");
            System.out.flush();
            String linie = tastatura.readLine();
            try {
                timp = Integer.parseInt(linie);
            }
        }
    }
}

```

```

        catch (NumberFormatException ex) {
            System.err.println("Format de intreg gresit: " + ex);
            System.exit(3);
        }
    }
    catch (IOException e) {
        System.out.println("Intrare gresita de la tastatura: " + e);
        System.exit(2);
    }
}

public void afiseaza(String s, int v, int timpUtilizat, int
putere) {
    if (s.equals("tv")) {
        System.out.println("TV-ul are volumul " + v);
        System.out.println("si a fost utilizat timp de " +
            timpUtilizat + " minute.");
        System.out.println("Puterea totala a fost " + putere);
    }
    else
        if (s.equals("aspirator")) {
            System.out.println("Aspiratorul are viteza " + v);
            System.out.println("si a fost utilizat timp de " +
                timpUtilizat + " minute.");
            System.out.println("Puterea totala a fost " + putere);
        }
}

public void calculeazaPutereConsumata(DispozitiveElectronice
dispozitiv) {
    int putere = dispozitiv.getTimpFolosinta() *
        dispozitiv.getPutere();
    totalPutere = totalPutere + putere;
}
}

```

Interfața `DispozitiveElectronice` declară metode referitoare la timpul utilizat de un dispozitiv electronic (aceste metode fiind de acces la atributele claselor implementate, s-a păstrat convenția de notare cu `set...` și `get...`) și la puterea curentului aparatului electronic pe unitatea de măsură. Clasa abstractă `Locuinta` (este abstractă, deoarece nu implementează metoda `schimbaVolum()`) se referă la situația pornirii și opririi unui aparat electronic. Clasele ne-abstracte `TV` și `Aspirator` extind clasa (de bază) `Locuinta` și implementează interfața `DispozitiveElectronice` prin definirea metodelor abstracte din acestea. Acestea au în comun atributul `volum` referitor la

volumul/viteza utilizat(ă) de dispozitivul electronic (televizor/aspirator). În plus, clasele TV și Aspirator au în particular un atribut timp. Chiar dacă un atribut provine de la superclasa Locuinta și unul este propriu fiecărei clase, instanțele celor două clase nu au legătură comună referitoare la acestea (adică modificarea valorii acestui atribut pentru obiectul clasei TV nu afectează valoarea aceluiași atribut pentru clasa Aspirator și reciproc). Deoarece s-a dorit utilizarea doar a obiectelor, și nu a metodelor statice, s-a creat un obiect al clasei AplicațieElectronice. Această clasă (de testare a aplicației noastre) are ca atribute timp, volum și, în plus față de clasele TV și Aspirator, un atribut totalPutere pentru a păstra valoarea puterii consumate de dispozitivul electronic curent pe perioada de timp utilizată. Se vor citi de la tastatură timpurile de întrebuințare a televizorului, respectiv aspiratorului, afișându-se volumul, timpul și puterea totală consumată. O posibilă execuție a programului de mai sus este:

```
Dati timpul (in minute): 60
TV-ul are volumul 2
si a fost utilizat timp de 60 minute.
Puterea totala a fost 6000
Dati timpul (in minute): 10
Aspiratorul are viteza 3
si a fost utilizat timp de 10 minute.
Puterea totala a fost 11000
```

3.4. Tablouri

Continuăm prezentarea tipurilor referință cu tipul *tablou*. În Java, *tablourile* sunt obiecte create dinamic și pot fi asigurate variabilelor de tip Object (toate metodele clasei Object pot fi apelate unui tablou). Un obiect tablou conține un număr de *variabile* (se mai numesc *elemente* sau *componente*), eventual zero elemente, caz în care se spune că tabloul este *vid*. Elementele unui tablou nu au nume, dar pot fi referențiate prin expresii de acces la tablou care se evaluează la valori întregi pozitive. Dacă un tablou are *n* elemente, spunem că *n* este lungimea tabloului (dat prin atributul length) și elementele pot fi accesate folosind indici de la 0 la *n*-1. Toate elementele unui tablou au același tip, numit *tipul componentelor* tabloului. Dacă tipul componentelor unui tablou este <Tip>, atunci tipul tabloului este <Tip> [].

Sintaxa generală a declarării unui tablou (fără crearea sau inițializarea acestuia) este:

```
<TipComponente> [][] ... [][] <Identificator> [][] ... [];
```

unde:

- <Identificator> reprezintă numele tabloului;
- există cel puțin o pereche de paranteze pătrate ([]) și nu are importanță unde apar, înainte sau după <Identificator> (de regulă se scriu înaintea identificatorului);

- <TipComponente> poate fi tip primitiv sau referință (sunt permise și tipurile interfață și clasă abstractă);
- tipul tabloului este <TipComponente> [][] ... [], unde numărul de [] este egal cu suma perechilor de paranteze pătrate care apar în declarație (numărul perechilor parantezelor se numește *dimensiunea* sau *adâncimea* tabloului).

Exemplul 2.7.4. evidențiază faptul că nu are importanță ordinea parantezelor. O variabilă de tip tablou memorează o referință de tip tablou. Declarația unei variabile de tip tablou nu creează întreg obiectul tablou sau alocă spațiu de memorie pentru componente, ci doar creează variabila care poate conține o referință la un tablou. Deoarece atributul length nu face parte din tipul tablou, o variabilă de tip tablou poate memora referințe la tablouri de diferite lungimi.

Un tablou se poate crea prin expresii de creare care folosesc operatorul new sau un inițializator de tablouri. O expresie de creare a unui tablou specifică tipul elementului, dimensiunea tabloului (numărul de perechi de paranteze pătrate) și lungimea tabloului cel puțin pentru prima pereche de paranteze pătrate. Numărul de elemente ale tabloului este dat de variabila instanță constantă numită length.

Sintaxa generală de creare a unui tablou folosind operatorul new este:

```
<Identificator> = new <Tip> [<Expresie1>][<Expresie2>] ...
[<ExpresieN>][][] ... []];
```

unde:

- <Identificator> este numele tabloului;
- <Tip> poate fi un tip primitiv sau referință (clasă (abstractă), interfață etc);
- <Expresie1>, <Expresie2>, ..., <ExpresieN> sunt de tip integral și se pot converti la compilare la tipul int, altfel apare o eroare la compilare.

O expresie de creare a unui tablou este evaluată la execuție astfel (în ordine):

1. sunt evaluate <Expresie1>, <Expresie2>, ..., <ExpresieN>. Dacă vreo evaluare se termină anormal, atunci expresiile următoare nu mai sunt evaluate și programul se termină anormal;
2. sunt verificate valorile expresiilor <Expresie1>, <Expresie2>, ..., <ExpresieN> și dacă vreuna este găsită negativă, atunci se aruncă o excepție de tip NegativeArraySizeException;
3. se alocă pentru tablou spațiu de memorie, iar dacă nu există spațiu suficient pentru alocarea respectivă, atunci expresia de creare a tabloului se termină anormal aruncând excepția OutOfMemoryError;
4. dacă N=1, atunci se creează un tablou uni-dimensional de lungime specificată și fiecare componentă a tabloului este inițializată cu valoarea sa implicită (Secțiunea 2.6);
5. dacă N>1, atunci se creează un tablou de dimensiune N (un tablou de tablouri).

Exemplul 3.4.1. Instrucțiunea de creare a tabloului:

```
int [][] a = new int[3][4];
```

este echivalentă cu:

```
int [][] a = new int[3][];
for (int i = 0; i < a.length; i++)
    a[i] = new int[4];
```

În acest caz, tabloul a va fi inițializat cu valoarea implicită a tipului int, adică 0.

Tablourile pot fi și neregulate, adică pot avea lungimi diferite pentru anumite dimensiuni. De exemplu, putem avea o matrice triunghiulară (adică alocăm memorie doar pentru elementele de sub diagonala principală).

Exemplul 3.4.2. Codul Java de mai jos:

```
int [][] a = new int[3][];
for (int i = 0; i < a.length; i++)
    a[i] = new int[i + 1];
```

va alocă memorie pentru tabloul triunghiular a. Astfel, vor exista elementele a[0][0], a[1][0], a[1][1], a[2][0], a[2][1], a[2][2].

Într-o expresie de creare a unui tablou, expresiile dintre paranteze se evaluează de la stânga la dreapta.

Exemplul 3.4.3. Codul Java de mai jos:

```
public class Test {
    public static void main(String[] args) {
        int i = 3;
        int [][] a = new int[i][i = 2];
        System.out.println("numar linii = " + a.length +
            ", numar coloane = " + a[0].length);
    }
}
```

va alocă memorie pentru tabloul a de dimensiuni [3][2], deoarece se evaluează întâi expresia i, apoi i = 2. Astfel, se va afișa:

```
numar linii = 3, numar coloane = 2
```

Un inițializator de tablouri furnizează valori inițiale pentru toate componentele sale (spre deosebire de C/C++ unde se puteau inițializa explicit doar primele valori, restul fiind completate de compilator cu valori implicite).

Sintaxa generală de creare a unui tablou folosind un inițializator de tablou este:

<Identificator> = <InițializatorTablou>;

iar <InițializatorTablou> are forma generală:

```
{<InițializatorVariabila1>[, <InițializatorVariabila2>]...
[, <InițializatorVariabilaN>]}
```

unde:

- <Identificator> este numele tabloului;
- <InițializatorVariabila1>, <InițializatorVariabila2>, ..., <InițializatorVariabilaN> pot fi <Expresie> sau <Inițializator Tablou>.

Lungimea tabloului construit va fi egală cu numărul de expresii. Fiecare expresie specifică o valoare pentru o componentă a tabloului și trebuie să fie compatibilă cu tipul componentei tabloului (în caz contrar, apare o eroare de compilare).

Exemplul 3.4.4. Codul Java de mai jos pune în evidență situația când se încearcă accesarea unei referințe null:

```
public class TestInițializareTablou {
    public static void main(String[] args) {
        int a[][] = { {3, 4}, null };
        for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
                System.out.println(a[i][j]);
    }
}
```

Acesta va afișa la execuție:

```
3
4
java.lang.NullPointerException
at TestInițializareTablou.main(TestInițializareTablou.java:6)
```

O componentă a unui tablou este accesată de o expresie de acces a unui tablou a cărui valoare este o referință la un tablou urmat de o expresie index încadrată de paranteze pătrate ([,]). Un tablou de lungime n poate fi indexat de întregii 0, 1, ..., n-1. Tablourile pot fi indexate de valori de tip int sau care pot fi convertite la int (de exemplu, valorile short, byte și char). În schimb, încercarea de accesare a unei componente tablou cu un index de tip long va genera o eroare la compilare. Accesul la tablouri este verificat în momentul execuției; orice încercare de utilizare a unui index mai mic decât 0 sau mai mare decât lungimea tabloului (dat prin atributul length) va implica aruncarea excepției IndexOutOfBoundsException.

Exemplul 3.4.5. Codul Java de mai jos pune în evidență situația când se încearcă accesarea unei componente cu un index prea mare:

```
class TestAccesTablou {
    public static void main(String[] args) {
        int[] v = new int[3];
        for (int i = 0; i < v.length; i++)
            v[i] = i;
```

```
System.out.println(v[v.length]);
```

Deoarece tabloul `v` este indexat de la 0 la `v.length-1`, programul de mai sus va afișa la execuție:

```
java.lang.ArrayIndexOutOfBoundsException
at TestAccesTablou.main(TestAccesTablou.java:6)
```

Membri unui tip tablou sunt:

1. atributul public final `length`, care conține numărul de componente ale tabloului;
2. metoda public `clone()`, care suprascrie metoda cu același nume din clasa `Object`;
3. toți membrii moșteniți din clasa `Object` (cu excepția lui `clone()` care este suprascrisă).

Exemplul 3.4.6. Codul Java de mai jos ilustrează crearea unei alte variabile de tip tablou care reprezintă o copie:

```
class TestClona {
    public static void main(String[] args) {
        int v1[] = {3, 4};
        int v2[] = (int[]) v1.clone();
        System.out.println(v1.hashCode() + " " + v2.hashCode());
        System.out.print((v1 == v2) + " ");
        v1[1]++;
        System.out.println(v2[1]);
    }
}
```

Variabilele de tip tablou `v1` și `v2` nu au aceeași adresă, deci modificarea ulterioară a „originalului” nu va implica și modificarea copiei. Pentru justificarea celor spuse vom apela metoda `hashCode()`, obținând astfel la execuția programului de mai sus:

```
7474923 3242435
false
```

O clonă a unui tablou multidimensional va conține doar adresa de bază distinctă și doar creează un nou tablou, subtablourile fiind comune cu tabloul inițial.

Exemplul 3.4.7. Codul Java de mai jos evidențiază crearea unei alte variabile de tip tablou bidimensional care reprezintă o copie:

```
class TestClonaTablouriBidimensionale {
    public static void main(String[] args) throws Throwable {
        int a[][] = {{1, 2}, {3, 4}};
```

```
int copie[][] = (int[][])a.clone();
a[0][1] = 5;
System.out.print(copie[0][1] + " ");
System.out.print((a == copie) + " ");
System.out.println(a[0] == copie[0] && a[1] == copie[1]);
}
```

Adresa de bază va fi distinctă, în schimb tablourile `a[0]`, `a[1]` sunt aceleași cu `copie[0]` și respectiv `copie[1]`. Modificarea originalului va implica și modificarea copiei și reciproc. Programul de mai sus va afișa la execuție:

```
5 false true
```

Fiecare tablou are un obiect `Class` asociat, care este comun tuturor tablourilor având componente de același tip și același număr de dimensiuni. Instanțele clasei `Class` reprezintă clasele și interfețele definite într-o aplicație Java.

Superclasa tipului tablou este clasa `Object`.

Exemplul 3.4.8. Codul Java de mai jos pune în evidență afișarea clasei și superclasei unui tablou unidimensional:

```
class TestSuperclasa {
    public static void main(String[] args) {
        int[] v = new int[10];
        System.out.println(v.getClass());
        System.out.println(v.getClass().getSuperclass());
    }
}
```

Ca și în Exemplul 2.6.1, la afișarea clasei curente se va folosi o paranteză pătrată urmată de litera `I`, care semnifică tipul `int`, superclasa tablourilor fiind clasa `Object`. Astfel, programul va afișa la execuție:

```
class [I
class java.lang.Object
```

Spre deosebire de alte limbaje de programare, un tablou de caractere nu este tot una cu un obiect `String`. Conținutul unui obiect `String` nu se poate schimba pe componente (eng. *immutable*), pe când elementele unui tablou se pot schimba pe componente. Vom vedea în capitolele ce urmează că există în clasa `String` metoda `toCharArray()` care returnează tabloul de caractere ce conține aceeași secvență de caractere ca și obiectul `String`. Clasa `StringBuffer` conține metode necesare pentru modificarea componentelor (eng. *mutable*).

Continuăm cu prezentarea transmiterii tablourilor ca parametri. Deoarece tablourile sunt de fapt niște obiecte, transmiterea lor ca parametri se realizează prin apel prin referință. Deci când un tablou se transmite ca parametru unei metode, se trimite o copie a adresei tabloului (se mai cheamă *referință a tabloului*). O referință nu este o

copie a tabloului, ci un pointer către tablou. Deci când o metodă schimbă parametrul formal de tip tablou, acesta schimbă și parametrul actual din apelul metodei.

Exemplul 3.4.9. Codul Java de mai jos pune în evidență transmiterea unui tablou unidimensional ca parametru pentru o metodă:

```
public class TransmitereTablouriParametri {
    public static void main(String args[]) {
        int [] a = new int[5];
        afiseazaSir(a);
        initializeazaSir(a);
        afiseazaSir(a);
    }
    static void initializeazaSir(int [] sir) {
        for(int i = 0; i < sir.length; i++)
            sir[i] = 1;
    }
    static void afiseazaSir(int [] sir) {
        for(int i = 0; i < sir.length; i++)
            System.out.print(sir[i] + " ");
        System.out.println();
    }
}
```

Inițial, tabloul `a` se inițializează de către sistem cu valori implicite (adică 0). Apelul metodei statice `afiseazaSir()` va transmite o copie a adresei tabloului `a`. Metoda `afiseazaSir()` nu va modifica valorile elementelor parametrului formal, adică ale tabloului `sir`, și deci nici ale tabloului `a`. În schimb, metoda `initializeazaSir()` va atribui literalul 1 ca valoare pentru componentele tabloului `sir`, deci și ale tabloului `a`. Astfel, programul Java de mai sus va afișa la execuție:

```
0 0 0 0 0
1 1 1 1 1
```

Exemplul 3.4.10. Codul Java de mai jos evidențiază faptul că la transmiterea unui tablou ca parametru se trimite o copie a adresei sale (și nu originalul):

```
public class TestAdresaTablouri {
    public static void main(String args[]) {
        int v[] = {2, 3};
        System.out.println("Înainte de apel, adresa lui v este " +
            v.hashCode());
        afiseazaTablou(v);
        f(v);
        System.out.println("După apel, adresa lui v este " +
            v.hashCode());
    }
}
```

```
afiseazaTablou(v);
}
public static void f(int [] sir) {
    int w[] = {5, 6};
    sir = w;
    System.out.println("În metoda f(), adresa lui sir este " +
        sir.hashCode());
    afiseazaTablou(sir);
}
static void afiseazaTablou(int [] sir) {
    System.out.print("Tabloul este: ");
    for(int i = 0; i < sir.length; i++)
        System.out.print(sir[i] + " ");
    System.out.println();
}
}
```

În metoda `main()`, tabloul `v[]` are o adresă în memorie (pe care o notăm cu A). În momentul apelului `f(v)`, se va transmite o copie a adresei lui A ca parametru pentru tabloul `sir`. Tabloul `w` va avea o altă adresă în memorie (pe care o notăm cu B). La asignarea `sir = w`, variabila `sir` va avea adresa B, astfel că modificările asupra lui `w` se vor vedea și în `sir`. Însă la terminarea execuției metodei `f()`, variabila `w` (deci și variabila `sir`) va fi inaccesibilă. Adresa tabloului `v` din memorie nu s-a schimbat, deci este tot A. Astfel, programul va afișa la execuție:

```
Înainte de apel, adresa lui v este 7474923
Tabloul este: 2 3
În metoda f(), adresa lui sir este 3242435
Tabloul este: 5 6
După apel, adresa lui v este 7474923
Tabloul este: 2 3
```

Exemplul 3.4.11. Codul Java de mai jos ilustrează transmiterea (și inițializarea) tablourilor bidimensionale ca parametri:

```
public class MatriceParametri {
    public static void main(String args[]) {
        int [][] v1 = new int[2][3];
        initializeazaUnu(v1);
        afiseazaMatrice(v1);
        int [][] v2 = { {1, 3, -4}, {4, -6, 3} };
        afiseazaMatrice(v2);
    }
    static void afiseazaMatrice(int [][] a) {
        for (int i = 0; i < a.length; i++) {
```

```

    for (int j = 0; j < a[0].length; j++)
        System.out.print(a[i][j] + " ");
    System.out.println();
}
}

static void initializeazaUnu(int [][] a) {
    int numarLinii = a.length;
    int numarColoane = a[0].length;
    for (int linie = 0; linie < numarLinii; linie++)
        for (int coloana = 0; coloana < numarColoane; coloana++)
            a[linie][coloana] = 1;
}
}

```

Programul va afișa la execuție:

```

1 1 1
1 1 1
1 3 -4
4 -6 3

```

3.5. Conversii ale tipului referință

În această secțiune, ne vom referi la conversii ale tipurilor *clasă*, *interfață* și *tablou*. În Secțiunea 2.8 am prezentat conversiile identice, primitive (implicite și explicite) și cea implicită la *String*. Au rămas de prezentat conversiile de referință (implicite și explicite), precum și cele explicite la *String*.

3.5.1. Conversii implicite ale tipului referință

Asemenea conversii nu necesită acțiuni speciale în momentul execuției și deci nu aruncă excepții la execuție. Se semnalează încă de la compilarea programului Java dacă o referință se poate converti implicit la alt tip. Notând cu A tipul referință sursă (clasă, interfață sau tablou) și cu B tipul referință destinație, la care se verifică în faza de compilare conversia implicită, distingem următoarele situații posibile:

1. Dacă A este un tip clasă, atunci:
 - a) Dacă B este un tip clasă, atunci A trebuie să coincidă cu B sau A trebuie să fie subclasă a lui B, altfel apare o eroare de compilare;
 - b) Dacă B este un tip interfață, atunci A trebuie să implementeze interfața B, altfel apare o eroare de compilare;
 - c) Dacă B este un tip tablou, atunci apare o eroare de compilare.
2. Dacă A este un tip interfață, atunci:
 - a) Dacă B este un tip clasă, atunci B trebuie să fie clasa *Object*, altfel apare o eroare de compilare;

- b) Dacă B este un tip interfață, atunci A trebuie să coincidă cu B sau A trebuie să fie subinterfață a lui B, altfel apare o eroare de compilare;
 - c) Dacă B este un tip tablou, atunci apare o eroare de compilare.
3. Dacă A este un tip tablou (de forma AA[]), atunci:
 - a) Dacă B este un tip clasă, atunci B trebuie să fie clasa *Object*, altfel apare o eroare de compilare;
 - b) Dacă B este un tip interfață, atunci B trebuie să coincidă cu *Cloneable* (singura interfață implementată de tablouri), altfel apare o eroare de compilare;
 - c) Dacă B este un tip tablou (de forma BB[]), atunci are loc una din afirmațiile:
 - AA și BB reprezintă același tip primitiv;
 - AA și BB sunt tipuri referință și AA este asignabil lui BB;
 - Dacă primele două nu pot avea loc, atunci apare o eroare de compilare.

Ideea principală a conversiilor implicite este că acestea se realizează de la „detaliu” la „general”. De exemplu, dacă C2 este o clasă Java și C1 superclasa sa, se poate face conversie implicită de la un obiect din C2 la un obiect din C1, deoarece clasa C1 are mai puțini membri decât C2 (care moștenește toți membrii din C1).

Exemplul 3.5.1. Următorul program Java pune în evidență câteva din situațiile posibile discutate mai sus:

```

class C1 { int x; }
class C2 extends C1 { int y; }
interface I1 { void f(int x); }
class C3 extends C1 implements I1 {
    int x;
    public void f(int x) { this.x = x; }
}

public class ConversiiReferintaCompilare {
    public static void main(String[] args) {
        // Asignari de variabile de tip clasă
        C1 obiectUnu = new C1();
        // asignarea este posibilă deoarece C2 este subclasa a lui C1
        obiectUnu = new C2();
        // reciproc, asignarea nu este posibilă deoarece este necesară
        // o conversie explicită de forma (C2)
        C2 obiectDoi = obiectUnu;
        // Asignari de variabile de tip Object
        // asignarea este posibilă deoarece C1 este subclasa a lui
        // Object
        Object obiectTrei = obiectUnu;
        int[] a = new int[3];
        // asignarea este posibilă deoarece clasa tablourilor este
        // subclasa a lui Object
    }
}

```



```

Object obiectPatru = a;
// Asignari de variabile de tip interfata
C3 obiectCinci = new C3();
// asignarea este posibila deoarece clasa C3 implementeaza I1
I1 obiectSase = obiectCinci;
// Asignari de variabile de tip tablou
byte[] b = new byte[4];
// asignarea nu este posibila deoarece tablourile a si b
// nu sunt definite pe acelasi tip primitiv
a = b;
C2 [] obiectSapte = new C2[3];
// asignarea este posibila deoarece C2 este subclasa a lui C1
C1 [] obiectOpt = obiectSapte;
// asignarea nu este posibila fiind necesara o conversie
// explicita la (C2 [])
obiectSapte = obiectOpt;
}
}

```

Execuția acestui program va implica afișarea a trei erori de compilare, așa cum se menționează și în comentariile programului.

3.5.2. Conversii explicite ale tipului referință

Aceste conversii necesită un test suplimentar la momentul execuției pentru a afla dacă valoarea referinței actuale este o valoare legitimă a acestui tip nou, în caz contrar se aruncă o excepție `ClassCastException`. Notăm cu A tipul referință sursă (clasă, interfață sau tablou) și cu B tipul referință destinație. Sintactic, conversia explicită necesită scrierea lui B între paranteze rotunde (adică (B)). O conversie explicită poate fi determinată ca fiind corectă încă de la faza de compilare. O conversie de la A la B este corectă la faza de compilare dacă și numai dacă A poate fi convertit la B printr-o conversie de asignare (Secțiunea 2.8.1).

Prezentăm mai întâi situațiile de conversie explicită care se verifică în faza de compilare:

1. Dacă A este un tip clasă, atunci:
 - a) Dacă B este un tip clasă, atunci A trebuie să coincidă cu B sau A trebuie să fie subclasă a lui B, sau B trebuie să fie subclasă a lui A, altfel apare o eroare de compilare;
 - b) Dacă B este un tip interfață, atunci:
 - i. Dacă A este o clasă nedeclarată `final`, atunci conversia explicită este corectă la compilare;
 - ii. Dacă A este o clasă declarată `final`, atunci A trebuie să implementeze B, altfel apare o eroare la compilare;

- c) Dacă B este un tip tablou, atunci A trebuie să fie clasa `Object`, altfel apare o eroare de compilare.
2. Dacă A este un tip interfață, atunci:
 - a) Dacă B este o clasă nedeclarată `final`, atunci conversia explicită este corectă la compilare;
 - b) Dacă B este o clasă declarată `final`, atunci B trebuie să implementeze A, altfel apare o eroare la compilare;
 - c) Dacă B este o interfață și A, B conțin metode cu aceeași semnătură, dar tipuri diferite returnate, atunci apare o eroare de compilare.
3. Dacă A este un tip tablou (de forma `AA[]`), atunci:
 - a) Dacă B este un tip clasă, atunci B trebuie să fie clasa `Object`, altfel apare o eroare de compilare;
 - b) Dacă B este un tip interfață, atunci B trebuie să coincidă cu `Cloneable` (singura interfață implementată de tablouri), altfel apare o eroare de compilare;
 - c) Dacă B este un tip tablou (de forma `BB[]`), atunci are loc una din afirmațiile:
 - i. AA și BB reprezintă același tip primitiv;
 - ii. AA și BB sunt tipuri referință și AA este convertit (implicit) la BB;
 - iii. Dacă ii. și iii. nu pot avea loc, atunci apare o eroare de compilare.

Exemplul 3.5.2. Următorul program Java pune în evidență câteva dintre situațiile posibile discutate mai sus:

```

class C1 { int x; }
class C2 extends C1 { int y; }
interface I1 { void f(int x); }
class C3 extends C1 implements I1 {
    int x;
    public void f(int x) { this.x = x; }
}

public class ConversiiReferintaCompilareExplicite {
    public static void main(String[] args) {
        // Asignari de variabile de tip clasa
        C1 obiectUnu = new C1();
        C2 obiectDoi = new C2();
        // asignarea este posibila deoarece C2 este subclasa a lui C1
        obiectUnu = obiectDoi;
        // reciproc, asignarea este posibila deoarece este necesara
        // o conversie explicita de forma (C2)
        obiectDoi = (C2) obiectUnu;
        // Asignari de variabile de tip interfata
        C3 obiectTrei = new C3();
        // asignarea este posibila la compilare deoarece este
    }
}

```

```
// facuta o conversie explicita. Totusi, depinde de tipul
// de la executie a lui obiectUnu. Este necesara o
// verificare la executie a tipului lui obiectUnu.
// obiectUnu poate sa nu apartina nici lui
// C3 nici unei subclase ale acestuia.
obiectTrei = (C3) obiectUnu;
// asignarea este posibila deoarece clasa C3 implementeaza I1
I1 obiectPatru = obiectTrei;
// La fel ca mai sus, asignarea este posibila
// la asignare, insa nu mai este posibila
// la executie, deoarece obiectUnu
// poate sa nu implementeze interfata I1
obiectPatru = (I1) obiectUnu;
}
```

Programul este corect la compilare, însă la execuție va arunca excepția `ClassCastException`:

```
Exception in thread "main" java.lang.ClassCastException: C2
at ConversiiReferintaCompilareExplicite.main (Conversii
ReferintaCompilareExplicite.java:26)
```

Exemplul 3.5.3. Următorul program Java pune în evidență situațiile de conversie referitoare la tablouri:

```
class C1 { int x; }
class C2 extends C1 { int y; }

public class ConversiiTablouriCompilareExplicite {
    public static void main(String[] args) {
        // Asignari de variabile de tip tablou
        int [] a = new int[5];
        // asignarea este posibila deoarece int[] este subclasa a
        // lui Object
        Object o = a;
        // Valoarea lui a nu se poate asigna lui b, deoarece a
        // este tablou cu componente de tip primitiv, pe cand
        // Integer nu este tip primitiv.
        // Se obtine eroare la compilare.
        Integer [] b = a;
        // Valoarea lui a nu se poate asigna lui c, deoarece a si
        // c sunt tablouri cu componente de tipuri diferite.
        // Se obtine eroare
        // la compilare.
```

```
short [] c = a;
C1 [] obiectUnu = new C1[10];
C2 [] obiectDoi = new C2[11];
// asignarea este corecta deoarece C2 este subclasa a lui C1
obiectUnu = obiectDoi;
// asignarea este corecta la faza de compilare, dar va arunca
// o exceptie ArrayStoreException la executie
obiectUnu[0] = new C1();
// asignarea nu este posibila, se obtine eroare la compilare
obiectDoi = obiectUnu;
}
```

Așa cum s-a menționat și în comentariile programului, sunt trei erori la compilare și o eroare la execuție.

Continuăm cu prezentarea situațiilor de conversie explicită care se verifică în faza de execuție, dar care n-au fost detectate (n-au putut fi verificate) la compilare. Anumite conversii explicite necesită o verificare la momentul execuției. Notăm cu A tipul referință sursă, cu B tipul referință destinație și cu C clasa obiectului referit de valoarea referinței la execuție. Dacă valoarea de la execuție este null, atunci este permisă conversia. Altfel, conversia explicită trebuie să verifice la execuție dacă tipul obiectului (clasa C) este compatibil cu B la asignare folosind algoritmul de mai sus (cel aplicat la faza de compilare). Iată situațiile de conversie explicită care se verifică în faza de execuție:

1. Dacă C este un tip clasă obișnuită (deci nu o clasă tablou), atunci:
 - a) Dacă B este un tip clasă, atunci C trebuie să coincidă cu B sau C trebuie să fie subclasă a lui B, altfel se aruncă o excepție de tip `ClassCastException` la execuție;
 - b) Dacă B este un tip interfață, atunci C trebuie să implementeze B, altfel se aruncă o excepție de tip `ClassCastException` la execuție;
 - c) Dacă B este un tip tablou, atunci se aruncă o excepție de tip `ClassCastException` la execuție.
2. Dacă C este un tip interfață, atunci:
 - a) Dacă B este o clasă, atunci B trebuie să fie clasa `Object`, altfel se aruncă o excepție de tip `ClassCastException` la execuție;
 - b) Dacă B este o interfață, atunci C trebuie să fie aceeași interfață cu B sau o subinterfață a lui B, altfel se aruncă o excepție de tip `ClassCastException` la execuție;
 - c) Dacă B este un tip tablou, atunci se aruncă o excepție de tip `ClassCastException` la execuție.
3. Dacă C este un tip tablou (de forma `CC[]`), atunci:
 - a) Dacă B este o clasă, atunci B trebuie să fie clasa `Object`, altfel se aruncă o excepție de tip `ClassCastException` la execuție;

- b) Dacă B este un tip interfață, atunci B trebuie să coincidă cu Cloneable (singura interfață implementată de tablouri), altfel se aruncă o excepție de tip ClassCastException la execuție;
- c) Dacă B este un tip tablou (de forma BB[]), atunci are loc una din afirmațiile:
 - i. BB și CC reprezintă același tip primitiv;
 - ii. BB și CC sunt tipuri referință și CC este convertit (implicit) la BB;
 - iii. Dacă ii. și iii. nu pot avea loc, atunci se aruncă o excepție de tip ClassCastException la execuție.

Exemplul 3.5.4. Următorul program Java ilustrează unele situații de conversie care se verifică la execuție (liniile sunt numerotate):

```

1. class C1 { int x; }
2. interface I1 { void f(int x); }
3. class C2 extends C1 implements I1 {
4.     int x;
5.     public void f(int x) { this.x = x; }
6. }
7. public class TestConversiei {
8.     public static void main(String[] args) {
9.         C1 obiectUnu = new C1();
10.        C2 obiectDoi = new C2();
11.        I1 obiectTrei;
12.        obiectDoi = (C2)obiectUnu;
13.        obiectUnu = obiectDoi;
14.        obiectTrei = (I1)obiectUnu;
15.        Long x = (Long)obiectUnu;
16.    }
17.}

```

Linia 12 va genera eroare la execuție, deoarece variabila obiectUnu poate să nu aparțină lui C2 sau unei clase derivate. Linia 14 este corectă și la compilare și la execuție, deoarece clasa C2 implementează interfața I1 (dacă ar fi lipsit linia 13, am fi obținut eroare la execuție). Linia 15 va afișa o eroare la compilare, fiindcă tipurile Long și C2 (și C1) nu au nimic în comun, deci nu se poate face nici conversie explicită.

Exemplul 3.5.5. Următorul cod Java evidențiază situațiile de conversie care se verifică la execuție (liniile sunt numerotate):

```

1. class C1 {
2.     int x = 1;
3.     C1 (int x) { this.x = x; }
4. }
5. class C2 extends C1 {
6.     int x = 3, y = 4;
7.     C2 (int x, int y) { super(x); this.y = y; }

```

```

8. }
9. public class TestConversie2 {
10.     public static void main(String args[]) {
11.         C1[] tablouUnu = new C2[5];
12.         for (int i = 0; i < tablouUnu.length; i++)
13.             tablouUnu[i] = new C2(i * 2, i * 3);
14.         C2[] tablouDoi = (C2[]) tablouUnu;
15.         for (int i = 0; i < tablouUnu.length; i++)
16.             System.out.print(tablouDoi[i].x + " " + tablouDoi[i].y
17.                               + " ");
18.    }

```

Linia 11 definește variabila tablouUnu ca având tipul la compilare C1[], iar la execuție C2[]. Linia 14 specifică variabila tablouDoi ca având tipurile de compilare și execuție C2[] (datorită conversiei explicite). Dacă ar fi lipsit conversia explicită (adică (C2[])), este clar că s-ar fi obținut eroare la compilare la linia 14 (conform regulilor de mai sus). Astfel, programul de mai sus este corect și va afișa la execuție:

```
3 0 3 3 3 6 3 9 3 12
```

Presupunem acum că linia 11 ar fi înlocuită cu:

```
11.    C1[] tablouUnu = new C1[5];
```

Atunci variabila tablouUnu are tipul la compilare și execuție C1[]. Programul este corect la compilare, însă la execuție va arunca o excepție de forma ClassCastException la linia 14, deoarece se aplică algoritmul de conversie discutat mai sus (punctul 3.c.ii).

Continuăm cu prezentarea accesului atributelor și metodelor la clasa curentă și la superclasele acesteia folosind conversii explicite. Astfel, fie C1 clasa de bază și C2 o clasă derivată a lui C1. Știm că se poate defini un obiect din clasa C1 care să fie instanță pentru C2, astfel:

```
C1 obiect = new C2();
```

Pentru rezolvarea ambiguității (eventuale a) membrilor suprascriși ai lui obiect, în mod implicit, atributele lui obiect sunt cele din C1, iar metodele lui obiect sunt cele din C2. Accesarea atributelor lui obiect din clasa C2 se face prin conversie explicită. Exemplele 3.2.26, 3.2.27 și 3.2.36 pun în evidență utilizarea și a altor procedee (prin nume calificat, întrebuițând cuvântul rezervat super).

Exemplul 3.5.6. Programul Java următor ilustrează situațiile de mai sus:

```

class C1 {
    int x = 1;
    void f(int x) {
        System.out.print("1 ");
    }

```

```

        this.x = x;
    }
}
class C2 extends C1 {
    int x = 2;
    public void f(int x) {
        System.out.print("2 ");
        this.x = x;
    }
}

public class TestMic {
    public static void main(String[] args) {
        C1 obiectUnu = new C2();
        System.out.print(obiectUnu.x + " ");
        obiectUnu.f(4);
        System.out.print(obiectUnu.x + " " + ((C2)obiectUnu).x + " ");
    }
}

```

Variabila `obiectUnu.x` se va referi la atributul `x` din clasa `C1`, iar `obiectUnu.f()` la metoda `f()` din clasa `C2`. Obținerea atributului `x` din clasa `C2` se poate face prin conversie explicită, astfel `((C2)obiectUnu).x`. Programul va afișa la execuție:

```
1 2 1 4
```

3.6. Concluzii

Capitolul de față s-a referit la prezentarea tipului referință, mai precis: clase, interfețe și tablouri. Secțiunea 3.2 descrie tipul clasă și anume: sintaxă, domeniul de vizibilitate, modificatorii clasei, clase abstracte și derivate, declarațiile membrilor (atribute și metode), supraîncărcare și suprascriere, niveluri de acces, inițializatori statici, declarațiile constructorilor, clase interioare, distrugerea obiectelor și eliberarea memoriei și împărțirea unei aplicații în mai multe fișiere sursă. Secțiunea 3.3 se referă la tipul interfață și anume: sintaxă, atribute, metode, moștenire multiplă, ascundere, suprascriere și supraîncărcare. Secțiunea 3.4 prezintă tablourile, acestora descriindu-li-se sintaxa, crearea, inițializarea și transmiterea către metode. Secțiunea 3.5 prezintă conversiile tipului referință (rămase „restante” din capitolul 2, unde s-au ilustrat conversiile tipurilor primitive). Sunt prezentate atât conversiile implicite/explicite, cât și cele de compilare/ execuție.

Încheiem concluziile Capitolului 3 cu asemănările și deosebirile dintre interfețe și clase abstracte:

1. O clasă abstractă poate implementa una sau mai multe metode. O interfață nu implementează nici o metodă.

2. O clasă poate implementa mai mult de o interfață, dar poate extinde doar (cel mult) o clasă abstractă.
3. O interfață este utilizată de compilator chiar la compilare pentru verificarea tipurilor. Dimpotrivă, o clasă abstractă implică moștenire, în care determinarea metodei adecvate din mulțimea metodelor supraîncărcate se realizează la momentul execuției.
4. O clasă abstractă implică moștenirea metodelor deja definite și definirea celor abstracte (în vederea creării de obiecte); o interfață specifică doar scheletul unei clase, fără posibilitatea moștenirii ulterioare.

3.7. Test grilă

Întrebarea 3.7.1. Fie următorul program Java:

```

class C1 {
    int x = 1;
    C1() {
        System.out.print("x = " + x);
    }
}
class C2 extends C1 {
    int y = 3;
    C2(int y) {
        this.y = y;
    }
}
public class Test {
    public static void main(String[] args) {
        C2 obiect = new C2();
        System.out.println(" y = " + obiect.y);
    }
}

```

Ce puteți spune despre acesta?

- a) Eroare la compilare: nu există constructor fără argumente.
- b) Programul este corect și la execuție va afișa: `y = 3`.
- c) Programul este corect și la execuție va afișa: `x = 1 y = 3`.

Întrebarea 3.7.2. Fie următorul program Java:

```

class C0 {
    int x = 1;
}
class C1 {

```

```

    int y = 2;
}
class C2 extends C0, C1 {
    int z = 3;
}
public class TestDoi {
    public static void main(String[] args) {
        C2 obiect = new C2();
        System.out.println(obiect.x + obiect.y + obiect.z);
    }
}

```

Ce puteți spune despre acesta?

- Eroare la compilare: clasa C2 nu poate fi derivată direct din două clase.
- Eroare la compilare: clasa C2 nu are constructor fără argumente.
- Programul este corect și la execuție va afișa: 6.

Întrebarea 3.7.3. Fie următorul program Java:

```

class x {
    private int x = 1;
    void x() {
        System.out.print(x + " ");
    }
}
class y extends x {
    private int x = 2;
    void x() {
        super.x();
        System.out.print(x);
    }
}
public class TestAtribute {
    public static void main(String[] args) {
        y obiect = new y();
        obiect.x();
    }
}

```

Ce puteți spune despre acesta?

- Eroare la compilare: nu se pot defini un atribut și o metodă cu același nume (x).
- Eroare la compilare: nu se poate defini un atribut cu numele clasei (x).
- Eroare la compilare: nu se poate defini un atribut cu același nume într-o clasă derivată (x).
- Programul este corect și la execuție va afișa: 1 2.

Întrebarea 3.7.4. Ce se va afișa la execuția următorului program?

```

public class TestStatic {
    public static void main(String args[]) {
        Exemplu a = new Exemplu();
        Exemplu b = new Exemplu();
        System.out.print("a.x = " + a.x);
        a.x = 100; b.x = 200;
        System.out.print(" a.x = " + a.x);
    }
}
class Exemplu {
    static int x = 0;
    Exemplu() {x++;};
}

```

- a.x = 2 a.x = 200
- a.x = 0 a.x = 100
- a.x = 1 a.x = 100
- a.x = 1 a.x = 101
- a.x = 2 a.x = 100

Întrebarea 3.7.5. Ce puteți spune despre următorul program?

```

public class Test {
    int i = 3, j = 4;
    public static void main(String [] arg) {
        System.out.println(i ++ + ++ j);
    }
}

```

- Va apărea eroare la compilare, deoarece parametrul metodei main() trebuie să fie String args [].
- Va apărea eroare la compilare, deoarece nu s-au inserat paranteze între operatorii ++, adică (i++) și (++j).
- Va apărea eroare la compilare, deoarece din funcția statică main() nu putem accesa variabilele nestatice i și j.
- La execuție se va afișa 8.
- La execuție se va afișa 9.

Întrebarea 3.7.6. Ce puteți afirma despre următorul program?

```

interface I1 { float x = 2.3f; }
public class Test implements I1 {
    public static void main(String[] args) {
        System.out.print(x + " ");
    }
}

```

```

    x = 6.7f;
    System.out.println(x);
}

```

- La apărerea eroare la compilare, deoarece valoarea variabilei x nu se mai poate modifica.
- La execuție se va afișa 2.3f 6.7f.
- La execuție se va afișa 2.3f 2.3f.
- La execuție se va afișa 2.3 6.7.
- La execuție se va afișa 2.3 2.3.

Întrebarea 3.7.7. Ce puteți spune despre următorul program Java?

```

interface I1 { int x = 2; }
interface I2 extends I1 { int y = 3; }
class C1 { int y = 4; }
public class C3 extends C1 implements I2 {
    public static void main(String args[]) {
        System.out.println("x = " + x + ", y = " + y);
    }
}

```

- La apărerea eroare la compilare, deoarece atributul y este ambiguu pentru C1 și I2.
- La execuție se va afișa x = 2, y = 3.
- La execuție se va afișa x = 2, y = 4.

Întrebarea 3.7.8. Ce puteți afirma despre următorul program Java?

```

class C1 {
    public String f() {
        return this.getClass().getName();
    }
}
abstract class C2 extends C1 {
    int x = 2;
    public abstract String f();
}
class C3 extends C2 {
    int y = 3;
    public String f() {
        return super.f() + ", x = " + x + ", y = " + y;
    }
}
public class TestAbstract {

```

```

public static void main(String args[]) {
    C3 obiect = new C3();
    System.out.println(obiect.f());
}

```

- Programul este corect și va afișa la execuție: C3, x = 2, y = 3.
- Programul este corect și va afișa la execuție: C1, x = 2, y = 3.
- La apărerea eroare la compilare, deoarece în clasa C2 s-a suprascris metoda f() ne-abstractă cu o metodă abstractă.
- La apărerea eroare la compilare, deoarece apelul super.f() din clasa C3 se referă la o metodă abstractă.
- La apărerea eroare la execuție, deoarece se intră într-o recursie infinită în metoda f() din clasa C3.

Întrebarea 3.7.9. Ce modificare trebuie făcută următorului program Java pentru a fi corect?

```

abstract class C1 {
    int x = 2;
    public final abstract String f();
}
final class C2 extends C1 {
    int y = 3;
    public String f() {
        return "x = " + x + ", y = " + y;
    }
}
public class TestFinal {
    public static void main(String args[]) {
        C2 obiect = new C2();
        System.out.println(obiect.f());
    }
}

```

- Trebuie șters cuvântul final din definiția metodei f() a clasei C1.
- Trebuie șters cuvântul final din definiția clasei C2.
- Trebuie șters cuvântul abstract din definiția clasei C1.
- Trebuie ștearsă inițializarea atributului x din declarația int x = 2; a clasei C1.
- Programul este corect și la execuție va afișa x = 2, y = 3.

Întrebarea 3.7.10. Ce puteți spune despre următorul program Java?

```

class C1 {
    int x = 1;
    void f(int x) { this.x = x; }
}

```

```

    int getX_C1() { return x; }
}
class C2 extends C1 {
    float x = 5.0f;
    int f(int x) { super.f((int)x); }
    float getX_C2() { return x; }
}
public class SuprascriereSiAscundere {
    public static void main(String args[]) {
        C2 obiect = new C2();
        obiect.f(4);
        System.out.print(obiect.getX_C2() + " ");
        System.out.println(obiect.getX_C1());
    }
}

```

- Programul este corect și va afișa la execuție: 5.0 4.
- Programul este corect și va afișa la execuție: 5 4.
- Programul este corect și va afișa la execuție: 4.0 4.
- Va apărea eroare la compilare, deoarece în clasa C2 s-a suprascris greșit atributul x din clasa C1.
- Va apărea eroare la compilare, deoarece metoda suprascrisă f() din clasa C2 întoarce un tip diferit de void.

Întrebarea 3.7.11. Ce puteți afirma despre următorul program Java?

```

class C1 {
    static String f() { return "Mesajul Unu din C1"; }
    String g() { return "Mesajul Doi din C1"; }
}
class C2 extends C1 {
    static String f() { return "Mesajul Unu din C2"; }
    String g() { return "Mesajul Doi din C2"; }
}
public class Test {
    public static void main(String[] args) {
        C1 obiect = new C2();
        System.out.println(obiect.f() + ", " + obiect.g());
    }
}

```

- Programul este corect și va afișa la execuție: Mesajul Unu din C1, Mesajul Doi din C1.
- Programul este corect și va afișa la execuție: Mesajul Unu din C1, Mesajul Doi din C2.

- Programul este corect și va afișa la execuție: Mesajul Unu din C2, Mesajul Doi din C1.
- Programul este corect și va afișa la execuție: Mesajul Unu din C2, Mesajul Doi din C2.
- Va apărea eroare la compilare, deoarece în clasa Test variabila obiect nu aparține clasei C2.

Întrebarea 3.7.12. Ce puteți spune despre următorul program Java?

```

class C1 {
    double x = 5.0f;
    void f(int x) {
        f((double)x + 2);
    }
    void f(double x) {
        this.x = x;
    }
}
public class Supraincarcare {
    public static void main(String args[]) {
        C1 obiect = new C1();
        obiect.f(4.0);
        System.out.println(obiect.x);
    }
}

```

- Programul este corect și va afișa la execuție: 4.0.
- Programul este corect și va afișa la execuție: 5.0.
- Programul este corect și va afișa la execuție: 6.0.
- Va apărea eroare la compilare, deoarece există ambiguitate în alegerea metodei f().
- Va apărea eroare la execuție, deoarece metoda f(int) intră într-o recursie infinită.

Întrebarea 3.7.13. Cercetați corectitudinea următorului program Java:

```

class C1 {
    static int f() { return j; }
    static int i = f();
    static int j = 1;
}
public class TestStatic {
    public static void main(String[] args) {
        System.out.println(C1.i);
    }
}

```

- a) Programul este corect și va afișa la execuție: 0.
- b) Programul este corect și va afișa la execuție: 1.
- c) Va apărea eroare la compilare, deoarece nu există nici un obiect declarat al clasei C1.
- d) Va apărea eroare la compilare, deoarece membrii clasei C1 nu sunt într-o ordine corectă.
- e) Va apărea eroare la compilare, deoarece s-a omis declararea clasei C1 ca fiind static.

Întrebarea 3.7.14. Ce puteți afirma despre următorul program Java (liniile sunt numerotate)?

```

1. public class SubiectLicentaDoi {
2.     static int x = 10;
3.     static { x += 5; }
4.     public static void main(String args[]) {
5.         System.out.println("x = " + x);
6.     }
7.     static { x /= 5; }
8. }
```

- a) Liniile 3 și 7 nu se vor compila, deoarece lipsesc numele metodelor și tipurile returnate.
- b) Linia 7 nu se va compila, deoarece poate exista doar un inițializator static.
- c) Codul se compilează și execuția produce ieșirea x = 10.
- d) Codul se compilează și execuția produce ieșirea x = 15.
- e) Codul se compilează și execuția produce ieșirea x = 3.

Întrebarea 3.7.15. Ce se va afișa la execuția următorului program Java?

```

public class SupracriereSupraincarcare extends Baza {
    public void functie(int j) {System.out.print(" j = " + j);}
    public void functie(String s) {System.out.print(" s = " + s);}
    public static void main(String args[]) {
        Baza b1 = new Baza();
        Baza b2 = new SupracriereSupraincarcare();
        b1.functie(5);
        b2.functie(6);
    }
}

class Baza {
    public void functie(int i) {System.out.print(" i = " + i);}
}
```

- a) i = 5 i = 6.
- b) j = 5 j = 6.
- c) i = 5 j = 6.

- d) s = 5 s = 6.
- e) Eroare la compilare, deoarece definiția clasei Baza apare după definiția clasei SupracriereSupraincarcare.
- f) Eroare la compilare, deoarece lipsește cuvântul virtual din metoda functie()

Întrebarea 3.7.16. Ce puteți spune despre următorul program Java?

```

class C1 {
    static int x = 1;
    int y = 2;
    static C1() { x = 4; }
    C1(int y) {
        this.y = y + 1;
    }
}

public class Test {
    public static void main(String args[]) {
        System.out.print(C1.x + " ");
        C1 obiect = new C1(5);
        System.out.println(obiect.y);
    }
}
```

- a) Programul este corect și va afișa la execuție: 1 6.
- b) Programul este corect și va afișa la execuție: 4 6.
- c) Programul este corect și va afișa la execuție: 1 2.
- d) Programul este corect și va afișa la execuție: 4 2.
- e) Va apărea eroare la compilare, deoarece s-a declarat unul din constructorii clasei C1 ca fiind static.

Întrebarea 3.7.17. Ce puteți afirma despre următorul program Java?

```

class C1 {
    int x = 1;
    C1(int x) { this.x = x; }
}

class C2 extends C1 {
    int y = 3;
    C2(int x) {
        this(x, y);
    }
    C2(int x, int y) {
        super(x);
        this.y = y;
    }
}
```



```

    public String toString() {
        return "x = " + x + ", y = " + y;
    }
}

public class Test {
    public static void main(String args[]) {
        C2 obiectUnu = new C2(4);
        C2 obiectDoi = new C2(5, 6);
        System.out.print(obiectUnu + " ");
        System.out.println(obiectDoi);
    }
}

```

- Programul este corect și va afișa la execuție: x = 4, y = 3 x = 5, y = 6.
- Programul este corect și va afișa la execuție: x = 1, y = 3 x = 5, y = 6.
- Programul este corect și va afișa la execuție: x = 1, y = 3 x = 1, y = 6.
- Programul este corect și va afișa la execuție: x = 1, y = 3 x = 1, y = 3.
- Va apărea eroare la compilare, deoarece primul constructor al clasei C2 conține un apel explicit în care al doilea parametru este o variabilă instanță.

Întrebarea 3.7.18. Indicați dacă programul următor este corect și ceea ce se va afișa la execuția acestuia:

```

class C1 {
    int x = 1;
    C1(int x) { this.x = x; }
}

class C2 extends C1 {
    int y = 2;
    public String toString() {
        return "x = " + x + ", y = " + y;
    }
}

public class Test {
    public static void main(String args[]) {
        C2 obiectUnu = new C2();
        C2 obiectDoi = new C2(3);
        System.out.print(obiectUnu);
        System.out.print(obiectDoi);
    }
}

```

- Programul este corect și va afișa la execuție: x = 1, y = 2 x = 1, y = 2.
- Programul este corect și va afișa la execuție: x = 1, y = 2 x = 3, y = 2.
- Programul este corect și va afișa la execuție: x = 1, y = 3 x = 1, y = 3.

- Programul este corect și va afișa la execuție: x = 1, y = 3 x = 3, y = 2.
- Va apărea eroare la compilare, deoarece clasa C1 nu are constructor fără parametri.

Întrebarea 3.7.19. Stabiliți dacă programul următor este corect sau nu:

```

interface I1 {
    int j = 3;
    int [] i = {j + 1, j + 2};
}

interface I2 {
    int j = 5;
    int [] i = {j + 2, j + 4};
}

public class Test implements I1, I2 {
    public static void main(String[] args) {
        System.out.print(I1.j + " ");
        System.out.print(I2.i[1] + " ");
        System.out.print(I1.i[0]);
    }
}

```

- Programul este corect și va afișa la execuție: 3 9 4.
- Va apărea eroare la compilare, deoarece interfețele I1 și I2 conțin declarațiile aceluiași atribut.
- Va apărea eroare la compilare, deoarece în interfețele I1 și I2 atributul i conține referință la atributul j.
- Va apărea eroare la compilare, deoarece interfețele I1 și I2 conțin declarațiile unui atribut de tip tablou.
- Va apărea eroare la compilare, deoarece atributele interfețelor I1 și I2 nu se pot accesa dintr-o metodă statică.

Întrebarea 3.7.20. Ce puteți spune despre corectitudinea următorului program Java?

```

interface IO {
    int x;
}

interface I1 extends IO {
    int [] y = {x + 1, x + 2};
}

interface I2 extends IO {
    int [] y = {x + 3, x + 4};
}

interface I3 extends I1, I2 {
    int z = I1.y[0], t = I2.y[1];
}

```

```
public class TestAtribute implements I3 {
    public static void main(String[] args) {
        I0.x = 3;
        System.out.print(x + " ");
        System.out.print(z + " ");
        System.out.print(t);
    }
}
```

- Programul este corect și va afișa la execuție: 3 4 7.
- Va apărea eroare la compilare, deoarece interfețele I1 și I2 conțin declarațiile aceluiași atribut.
- Va apărea eroare la compilare, deoarece în interfața I0 atributul x nu este inițializat.
- Va apărea eroare la compilare, deoarece interfața I3 accesează un atribut ambiguu.
- Va apărea eroare la compilare, deoarece attributele interfețelor I0 și I3 nu se pot accesa dintr-o metodă statică.

Întrebarea 3.7.21. Ce puteți afirma despre următorul program Java?

```
class ExceptiaNoastraUnu extends Exception {
    ExceptiaNoastraUnu() { super(); }
}
class ExceptiaNoastraDoi extends Exception {
    ExceptiaNoastraDoi() { super(); }
}
interface I1 {
    int f(int x) throws ExceptiaNoastraUnu, ExceptiaNoastraDoi;
}
interface I2 extends I1 {
    int f(int x) throws ExceptiaNoastraDoi;
}
class C1 implements I1 {
    public int f(int x) throws ExceptiaNoastraUnu,
        ExceptiaNoastraDoi {
        if (x < 0) throw new ExceptiaNoastraUnu();
        return x;
    }
    public int f(int x) throws ExceptiaNoastraDoi {
        if (x < 0) throw new ExceptiaNoastraDoi();
        return x;
    }
}
public class TestSuprascriereMetode {
    public static void main(String[] args) {
```

```
C1 obiect = new C1();
try {
    System.out.print(obiect.f(2) + " ");
    System.out.print(obiect.f(-2));
}
catch (ExceptiaNoastraUnu e) {
    System.out.print("A aparut o exceptie: " + e);
}
catch (ExceptiaNoastraDoi e) {
    System.out.print("A aparut o exceptie: " + e);
}
}
```

- Programul este corect și va afișa la execuție: 2 A aparut o exceptie: ExceptiaNoastraUnu.
- Programul este corect și va afișa la execuție: 2 A aparut o exceptie: ExceptiaNoastraDoi.
- Va apărea eroare la compilare, deoarece metoda f() din interfața I2 intră în contradicție cu metoda f() din I1.
- Va apărea eroare la compilare, deoarece interfețele I1 și I2 nu conțin atributul x.
- Va apărea eroare la compilare, deoarece în clasa C1 nu se poate implementa metoda f() din interfața I1.

Întrebarea 3.7.22. Selectați varianta corectă referitoare la corectitudinea programului Java de mai jos:

```
interface I1 {
    int f(int x);
}
interface I2 extends I1 {
    float f(float x);
    double f(double x);
}
class C1 implements I2 {
    public int f(int x) { return x; }
    public float f(float x) { return x; }
    public double f(double x) { return x; }
}
public class TestSupraincarcareMetode {
    public static void main(String[] args) {
        C1 obiect = new C1();
        System.out.print(obiect.f(2) + " ");
        System.out.print(obiect.f(4.5) + " ");
```

```
System.out.print(obiect.f(2.5f));
```

- a) Programul este corect și va afișa la execuție: 2 4.5 2.5.
- b) Programul este corect și va afișa la execuție: 2 4.5 2.5f.
- c) Va apărea eroare la compilare, deoarece metodele f() din interfața I2 intră în contradicție cu metoda f() din I1.
- d) Va apărea eroare la compilare, deoarece metodele f() din interfața I2 intră în contradicție între ele.
- e) Va apărea eroare la compilare, deoarece metodele f() din interfețele I1 și I2 nu sunt declarate public.

Întrebarea 3.7.23. Cercetați corectitudinea următorului program:

```
public class Test {
    public static void main(String args[]) {
        C1 obiect = new C1();
        obiect.f(4, 3);
    }
}

class C1 {
    public void f(int xx, final int yy) {
        int a = xx + yy;
        final int b = xx - yy;
        class C2 {
            public void g() {
                System.out.print("a = " + a);
                System.out.print(", b = " + b);
            }
        }
        C2 obiectDoi = new C2();
        obiectDoi.g();
    }
}
```

- a) Programul este corect și va afișa la execuție: a = 7, b = 1.
- b) Programul este corect și va afișa la execuție: a = 4, b = 3.
- c) Va apărea eroare la compilare, deoarece din metoda g() nu putem accesa variabila locală a din metoda f().
- d) Va apărea eroare la compilare, deoarece clasa C2 nu poate fi definită în metoda f() din clasa C1.
- e) Va apărea eroare la compilare, deoarece nu se creează în clasa Test un obiect de tip C1.C2.

Întrebarea 3.7.24. Verificați corectitudinea următorului program Java:

```
public class Exemplu {
    int v1[] = {0}, v2[] = {1};
    public static void main(String args[]) {
        int v1[] = {2}, v2[] = {3};
        System.out.print(v1[0] + " " + v2[0] + " ");
        f(v1, v2);
        System.out.println(v1[0] + " " + v2[0]);
    }
    public static void f(int v1[], int v2[]) {
        int w[] = {4};
        v1 = w;
        v2[0] = w[0];
        System.out.print(v1[0] + " " + v2[0] + " ");
    }
}
```

- a) Programul este corect și va afișa la execuție: 0 1 4 4 0 1.
- b) Programul este corect și va afișa la execuție: 2 3 4 4 0 1.
- c) Programul este corect și va afișa la execuție: 2 3 4 4 2 4.
- d) Programul este corect și va afișa la execuție: 2 3 4 4 4 4.
- e) Va apărea eroare la compilare, deoarece atributele v1 și v2 sunt redefinite ca variabile locale în metoda main().

Întrebarea 3.7.25. Ce puteți spune despre următorul program Java (liniile sunt numerotate)?

```
1. class C1 { int x = 1; }
2. class C2 extends C1 { int y = 2; }
3. public class TestCompilare {
4.     public static void main(String[] args) {
5.         C1 obiectUnu = new C1();
6.         System.out.print(obiectUnu.x + " ");
7.         obiectUnu = new C2();
8.         System.out.print(obiectUnu.x + " ");
9.         C2 obiectDoi = (C2) obiectUnu;
10.        System.out.print(obiectDoi.x + " ");
11.    }
12. }
```

- a) Programul este corect și va afișa la execuție: 1 1 1.
- b) Va apărea eroare la compilare la linia 7, deoarece obiectUnu este deja definit la linia 5.
- c) Va apărea eroare la compilare la linia 9, deoarece obiectUnu este de tip C1 definit la linia 5.

- d) Vor apărea erori de compilare la liniile 5, 7 și 9, deoarece într-o metodă statică nu putem crea variabile obiect.
- e) Va apărea eroare la execuție, aruncând excepția `ClassCastException`.

Întrebarea 3.7.26. Stabiliți care afirmații sunt adevărate pentru programul Java din cele de mai jos (liniile sunt numerotate):

```

1. public class TestTablouri {
2.     public static void main(String[] args) {
3.         int[] a = {2, 6};
4.         Object b = a;
5.         System.out.print(b[0] + " ");
6.         byte[] c = new byte[2];
7.         c = a;
8.         System.out.print(c[1] + " ");
9.     }
10.)
```

- a) Programul este corect și va afișa la execuție: 2 6.
- b) Va apărea eroare la compilare doar la linia 4, deoarece b necesită o conversie explicită.
- c) Va apărea eroare la compilare doar la linia 5, deoarece b[0] nu poate fi accesat.
- d) Vor apărea erori de compilare la linia 5, deoarece b[0] nu poate fi accesat și la linia 7, deoarece tipurile componentelor tablourilor a și c nu sunt compatibile.
- e) Va apărea eroare la execuție din cauza liniei 5, aruncându-se excepția `ClassCastException`.

Întrebarea 3.7.27. Indicați dacă program Java de mai jos este corect sau stabiliți natura erorilor (liniile sunt numerotate):

```

1. class C1 { int x = 1; }
2. class C2 extends C1 { int y = 2; }
3. interface I1 { int x = 3; void f(int x); }
4. class C3 extends C1 implements I1 {
5.     int x = 4;
6.     public void f(int x) { this.x = x; }
7. }
8. public class TestConversii {
9.     public static void main(String[] args) {
10.         C3 obiectUnu = new C3();
11.         I1 obiectDoi = obiectUnu;
12.         System.out.print(obiectDoi.x);
13.         C2 [] obiectTrei = new C2[3];
14.         System.out.print(obiectTrei[0].x + " ");
15.         C1 [] obiectPatru = obiectTrei;
16.         System.out.print(obiectPatru[1].x + " ");
```

```

17.         obiectTrei = (C2[]) obiectPatru;
18.     }
19. }
```

- a) Programul este corect și va afișa la execuție: 3 2 1.
- b) Programul este corect la compilare și va afișa la execuție 3, urmat de aruncarea excepției `NullPointerException` la linia 14.
- c) Programul este corect la compilare și va afișa la execuție 3, urmat de aruncarea excepției `NullPointerException` la linia 16.
- d) Va apărea eroare la compilare doar la linia 15, deoarece tipurile componentelor tablourilor obiectTrei și obiectPatru nu sunt compatibile.
- e) Va apărea eroare la execuție la linia 17, aruncându-se excepția `ClassCastException`.

Întrebarea 3.7.28. Ce puteți spune despre următorul program Java?

```

class C1 { int x = 1; }
interface I1 { int x = 2; void f(int x); }
class C2 extends C1 implements I1 {
    int x = 3;
    public void f(int x) { this.x = x; }
}

public class Test {
    public static void main(String[] args) {
        C1 obiectUnu = new C2();
        System.out.print(obiectUnu.x);
        I1 obiectDoi = (C2) obiectUnu;
        System.out.println(" " + obiectDoi.x);
    }
}
```

- a) Programul este corect și va afișa la execuție: 1 2.
- b) Programul este corect și va afișa la execuție: 3 2.
- c) Programul este corect și va afișa la execuție: 3 3.
- d) Va apărea eroare de compilare la crearea lui obiectDoi.
- e) Va apărea eroare la execuție la afișarea lui obiectDoi.x, aruncându-se excepția `ClassCastException`.

Întrebarea 3.7.29. Verificați dacă programul de mai jos este corect:

```

interface I1 { int x = 1; void f(int x); }
class C1 implements I1 {
    int x = 2;
    public void f(int x) { this.x = x; }
}
```

```

public class Test {
    public static void main(String[] args) {
        C1 obiectUnu = new C1();
        System.out.print(obiectUnu.x);
        I1 obiectDoi = obiectUnu;
        System.out.println(" " + obiectDoi.x);
    }
}

```

- Programul este corect și va afișa la execuție: 2 1.
- Programul este corect și va afișa la execuție: 2 2.
- Va apărea eroare de compilare la crearea lui obiectDoi, fiind necesară o conversie explicită de forma (C2).
- Va apărea eroare de compilare la crearea lui obiectDoi, deoarece atributul x se suprascrive.
- Va apărea eroare la execuție la afișarea lui obiectDoi.x, aruncându-se excepția ClassCastException.

Întrebarea 3.7.30. Ce puteți afirma despre următorul program Java?

```

class C1 {
    int x = 1;
    void f(int x) {
        System.out.print("Unu ");
        this.x = x;
    }
}
interface I1 { int x = 2; void f(int x); }
class C2 extends C1 implements I1 {
    int x = 3;
    public void f(int x) {
        System.out.print("Doi ");
        this.x = x;
    }
}
public class Test {
    public static void main(String[] args) {
        C1 obiectUnu = new C2();
        System.out.print(obiectUnu.x + " ");
        obiectUnu.f(4);
        I1 obiectDoi = (C2) obiectUnu;
        obiectDoi.f(5);
        System.out.println(obiectDoi.x);
    }
}

```

```

}

```

- Programul este corect și va afișa la execuție: 1 Unu Doi 2.
- Programul este corect și va afișa la execuție: 1 Unu Doi 3.
- Programul este corect și va afișa la execuție: 1 Doi Doi 2.
- Programul este corect și va afișa la execuție: 3 Doi Doi 2.
- Programul este corect și va afișa la execuție: 3 Doi Doi 3.
- Va apărea eroare de compilare la crearea lui obiectDoi, deoarece atributul x se suprascrive.
- Va apărea eroare la execuție la afișarea lui obiectDoi.x, aruncându-se excepția ClassCastException.

Întrebarea 3.7.31. Indicați dacă programul de mai jos este corect și, în caz afirmativ, ce va apărea pe ecran:

```

class C1 {
    int x = 1;
    void f(int x) {
        System.out.print("1 ");
        this.x = x;
    }
}
class C2 extends C1 {
    int x = 2;
    public void f(int x) {
        System.out.print("2 ");
        this.x = x;
    }
}
class C3 extends C1 {
    int x = 3;
    public void f(int x) {
        System.out.print("3 ");
        this.x = x;
    }
}
public class Test {
    public static void main(String[] args) {
        C1 obiectUnu = new C2();
        C1 obiectDoi = new C3();
        System.out.print(obiectUnu.x + " " + obiectDoi.x + " ");
        obiectUnu.f(4);
        obiectDoi.f(5);
        System.out.print(obiectUnu.x + " " + ((C2)obiectUnu).x +
            " ");
    }
}

```

```
System.out.print(obiectDoi.x + " " + ((C3)obiectDoi).x);
}
```

- a) Programul este corect și va afișa la execuție: 1 1 1 1 1 4 1 5.
- b) Programul este corect și va afișa la execuție: 1 1 2 3 1 4 1 5.
- c) Programul este corect și va afișa la execuție: 2 3 2 3 2 4 3 5.
- d) Programul este corect și va afișa la execuție: 2 3 2 3 4 4 5 5.
- e) Vor apărea erori de compilare la crearea instanțelor `obiectUnu` și `obiectDoi`, deoarece atributul `x` se suprascrive.
- f) Va apărea eroare la execuție la afișarea lui `obiectUnu.x`, aruncându-se excepția `ClassCastException`.

Întrebarea 3.7.32. Ce puteți afirma despre programul Java de mai jos?

```
class C1 {
    int x = 1;
    C1 (int x) { this.x = x; }
}
class C2 extends C1 {
    int x = 3, y = 4;
    C2 (int x, int y) { super(x); this.y = y; }
}
public class Test {
    public static void main(String args[]) {
        C1[] tablouUnu = new C2[3];
        for (int i = 0; i < tablouUnu.length; i++)
            tablouUnu[i] = new C2(i * 2, i * 3);
        C2[] tablouDoi = (C2[]) tablouUnu;
        for (int i = 0; i < tablouUnu.length; i++)
            System.out.print(tablouDoi[i].x + " " + tablouDoi[i].y +
                " ");
    }
}
```

- a) Programul este corect și va afișa la execuție: 1 0 1 3 1 6.
- b) Programul este corect și va afișa la execuție: 1 4 1 4 1 4.
- c) Programul este corect și va afișa la execuție: 3 0 3 3 3 6.
- d) Programul este corect și va afișa la execuție: 3 4 3 4 3 4.
- e) Vor apărea erori de compilare la crearea instanțelor `tablouUnu` și `tablouDoi`, deoarece atributul `x` se suprascrive.
- f) Va apărea eroare la execuție la afișarea lui `tablouUnu.x`, aruncându-se excepția `ClassCastException`.

Întrebarea 3.7.33. Verificați corectitudinea programului următor și specificați ce se va afișa:

```
class C1 {
    int x = 1;
    C1 (int x) { this.x = x; }
}
class C2 extends C1 {
    int x = 2, y = 3;
    C2(int x) { super(x); }
    C2(int x, int y) { super(x); this.y = y; }
}
public class ApelExplicitConstructor {
    public static void main(String args[]) {
        C1 obiectUnu = new C1(4);
        C2 obiectDoi = new C2(5);
        C2 obiectTrei = new C2(6, 7);
        System.out.print(obiectUnu.x + " ");
        System.out.print(obiectDoi.x + " " + obiectDoi.y + " ");
        System.out.print(obiectTrei.x + " " + obiectTrei.y);
    }
}
```

- a) Programul este corect și va afișa la execuție: 4 1 2 1 7.
- b) Programul este corect și va afișa la execuție: 4 1 3 6 7.
- c) Programul este corect și va afișa la execuție: 4 2 3 2 7.
- d) Programul este corect și va afișa la execuție: 4 5 3 6 7.
- e) Vor apărea erori de compilare la crearea instanțelor `obiectDoi` și `obiectTrei`, deoarece atributul `x` se suprascrive.
- f) Va apărea eroare la execuție la afișarea lui `ObiectDoi.x`, aruncându-se excepția `ClassCastException`.

3.8. Exerciții propuse spre implementare

Exercițiul 3.8.1. Scrieți un program Java care descrie corpul numerelor complexe. Implementați constructori pentru inițializarea unui număr complex și metode pentru suma, produsul a două numere complexe și modulul unui număr complex. Acesta poate avea scheletul:

```
class Complex {
    private double re, im;
    public Complex();
    public Complex(double re);
    public Complex(double re, double im);
}
```

```

    public Complex suma(Complex z);
    public Complex produs(Complex z);
    public double modul();
    public String toString();
}

```

Exercițiul 3.8.2. Scrieți un program Java pentru simularea operațiunilor asupra unui cont bancar. Implementarea trebuie să conțină ca atribute numărul contului, tipul contului, numele și adresa clientului, suma de bani existentă și dobânda. Definiți constructori cu parametri pentru inițializarea numărului contului, tipului contului, numele și adresa clientului. Ca metode, descrieți depunerea unei sume de bani, retragerea unei sume de bani, aflarea sumei de bani curente (capital), aflarea dobânzii curente. Banca respectivă are două tipuri de conturi pentru persoane fizice: cu dobândă la vedere (cu posibilitate de operațiuni zilnice) și la termen (cu reținerea sumei de bani până la termenul stabilit – se mai numește cont de economii). Suma de bani dintr-un cont la vedere poate chiar scădea (există posibilitatea dobânzii negative), dar nu și suma de bani dintr-un cont la termen. La sfârșitul fiecărei luni/an, la suma inițială se adaugă suma de bani obținută din dobânda contului de la scadență. Folosind moștenirea, scrieți clase Java pentru descrierea fiecărui tip de cont bancar. Utilizând un atribut static, numărați câte persoane fizice au cont deschis la banca respectivă.

Exercițiul 3.8.3. Scrieți un program Java pentru lucrul cu figuri geometrice. Definiți o clasă de bază *Punct* și clasele derivate *Dreptunghi* și *Cerc*. Testați aplicația creând câteva instanțe și verificați metodele scrise. Cercul se reprezintă ca un punct (centrul cercului) și o rază, iar dreptunghiul, ca două puncte diagonal opuse. Aplicația poate avea scheletul:

```

class Punct {
    protected int x, y;
    public Punct(int x, int y);
    public muta(int dx, int dy);
    public String toString();
}

class Dreptunghi extends Punct {
    protected int x0, y0;
    public Dreptunghi(int x, int y, int x0, int y0);
    public muta(int dx, int dy);
    public String toString();
}

class Cerc extends Punct {
    protected int r;
    public Cerc(int x, int y, int r);
    public muta(int dx, int dy);
}

```

```

    public String toString();
}

```

Exercițiul 3.8.3. Scrieți un program Java care definește o clasă *DataCalendaristica*. Datele calendaristice se pot trimite în două formate ZZ/LL/AAAA (formatul românesc de scriere a datelor calendaristice are forma zi/lună/an, de exemplu 20/12/2002) sau 20 decembrie 2002. Clasa *DataCalendaristica* are doi constructori care inițializează ziua, luna și anul datei calendaristice. Scrieți și câte o metodă de afișare a unui obiect din clasa *DataCalendaristica* (supraîncărcată) pentru fiecare din aceste formate de date calendaristice.

Exercițiul 3.8.4. Implementați o clasă pentru numere raționale. Aceasta va avea ca atribute numitorul și numărătorul, trei constructori: implicit, cu un parametru și cu doi. Se va scrie câte o metodă pentru operațiile de adunare, scădere, înmulțire și împărțire. Tot timpul fracția reținută de un obiect trebuie să fie ireductibilă (să nu se mai poată simplifica). Se va crea și o clasă auxiliară pentru aflarea c.m.m.m.c și c.m.m.d.c.

Exercițiul 3.8.5. Scrieți o aplicație Java care să permită efectuarea de operații cu polinoame. Operațiile permise vor fi adunarea, scăderea, înmulțirea, împărțirea, c.m.m.d.c, c.m.m.m.c. și se va oferi posibilitatea aflării gradului unui polinom.

4. Șiruri și structuri dinamice de date

4.1. Cuvinte cheie

- șiruri de caractere
- clasele String, StringBuffer, StringTokenizer
- listă înlănțuită, arbori, arbori binari
- enumerator, tabelă de dispersie, mulțime, stivă
- proprietăți

4.2. Clasa String

Reamintim faptul că tipul `char` este un tip primitiv, care memorează un caracter reprezentat pe 16 biți. Am văzut că putem defini și lucra cu șiruri de caractere. Însă Java pune la dispoziție tipul `String`, având astfel o manieră elegantă de lucru cu șirurile de caractere. `String`-urile sunt obiecte din clasa `String`.

Un literal `String` este o secvență de caractere cuprinsă între ghilimele (de exemplu: "Acesta este un literal"). Dacă se dorește afișarea ghilimelelor, atunci se folosește `\`, iar `\n` pentru trecerea la linie nouă ș.a.m.d.

Crearea unei variabile `String` se realizează exact ca și celelalte variabile.

Exemplul 4.2.1.

```
String sir = "Primul sir";
sir = "Al doilea sir";
```

Când se atribuie unei variabile de tip `String` o altă valoare, vechea referință se pierde, valoarea sa urmând a fi eliberată automat din memorie de către colectorul de gunoai. Se mai spune că obiectele `String` sunt fixe (nemodificabile).

La unele versiuni de Java, inițializarea șirurilor este obligatorie o dată cu declararea lor.

Exemplul 4.2.2.

```
String sir1;           // eroare
String sir2 = null;    // corect
String sir3 = "";      // corect
```

Dacă un șir se inițializează cu un literal `String`, atunci i se va atribui adresa acestuia din memorie. Acești literal `String` sunt păstrați în tabela de literali (eng. *literal pool*). Operatorul `==` aplicat pentru două `String`-uri testează dacă acestea pointează către aceeași adresă de memorie. În exemplul de mai jos, vom puncta această diferență sensibilă:

Exemplul 4.2.3.

```
String a1 = "abc"; // literalul "abc" are o adresa fixa din memorie
String b1 = "abc"; // a1 si b1 pointeaza la aceeaasi adresa de
                    // memorie

if (a1 == b1)
    System.out.println("Sirurile pointeaza la aceeaasi adresa");
// adevarat
else
    System.out.println("Sirurile pointeaza la adrese diferite");

String a2 = "abc";
String b2 = new String("abc"); // b2 va pointa la alta adresa
                               // de memorie fata de a2
```

```
if (a2 == b2)
    System.out.println("Sirurile pointeaza la aceeaasi adresa");
else // adevarat
    System.out.println("Sirurile pointeaza la adrese diferite");
```

Utilizarea operatorului `new` pentru un `String` conduce la crearea unui spațiu din memoria program în momentul execuției programului, și nu în tabela de literali. Putem stabili ca un `String` creat cu `new` să fie plasat în tabela de literali pentru o reutilizare ulterioară sau pentru re folosirea unui `String` identic din tabela de literali. Astfel putem utiliza metoda `intern()` din clasa `String`. În programele care folosesc multe `String`-uri similare, acest procedeu poate reduce spațiul de memorie utilizat. Mai mult, în programele care necesită comparații de `String`-uri care sunt memorate în tabela de literali, putem utiliza operatorul `==` (care verifică egalitatea referințelor), în loc de metoda `equals()` care este mult mai lentă, deoarece verifică egalitatea caracter cu caracter.

Exemplul 4.2.4.

```
String a1 = new String("abc");
String b1 = new String("abc");
String a2 = a1.intern(); // a2 se afla in tabela de literali
String b2 = b1.intern(); // b2 se afla in tabela de literali
// a2 si b2 pointeaza la aceeaasi adresa de memorie
if (a2 == b2) // adevarat
    System.out.println("Sirurile pointeaza la aceeaasi adresa");
else
    System.out.println("Sirurile pointeaza la adrese diferite");
```

Din moment ce variabilele `String` sunt obiecte, putem crea șiruri de `String`-uri.

Exemplul 4.2.5. Crearea șirurilor de `String`-uri.

```
String[] nume = new String[5];
nume[0] = "Goldan";
nume[1] = "Vasile";
```

Amintim și faptul că funcția `main()` are ca argument un șir de `String`-uri, permițând astfel primirea de argumente de la linia de comandă.

Operatorul `+` concatenează două (sau mai multe) șiruri de caractere. Operatorul `+=` creează un nou `String`, adăugând șirul din partea dreaptă a operatorului.

Exemplul 4.2.6.

```
String sir1 = "abc" + "def";
String sir2 = "abc";
sir2 += "def";
System.out.println("sir2 = " + sir2); // raspuns: sir2 = abcdef
```

În exemplul de mai sus, putem spune că `String`-urile sunt nemodificabile? Da. În realitate, se mai creează o zonă de memorie necesară pentru memorarea noului cuvânt ("abcdef") și vechea valoare ("abc") se pierde.

Operatorul `+` se asociază de la stânga la dreapta. Dacă un operand este `String`, atunci se convertește celălalt operand la `String`, tipul rezultatului întors fiind `String`.

Exemplul 4.2.7. Următoarea secvență de cod va afișa mesajul cincizeci si cinci = 55 10 = zece.

```
String sir3 = "cincizeci si cinci = " + 5 + 5;
String sir4 = 5 + 5 + " = zece ";
System.out.println(sir3 + " " + sir4);
```

Una dintre cele mai importante metode de conversie dintre tipuri arbitrare și `String` este metoda statică `toString()`, care posedă numeroase variante supraîncărcate. Există, de asemenea, un număr de metode de convertire a șirurilor de caractere într-o varietate de tipuri.

Am văzut că tipurile predefinite `byte`, `short`, `int`, `float`, `double`, `long`, `boolean`, chiar nu necesită crearea instanțelor cu operatorul `new`. Pentru uniformizare, există clasele „wrapper” asociate tipurilor predefinite: `Integer`, `Float`, `Double`, `Long`, `Boolean`, `Character`. De exemplu, când se folosește operatorul `+`, Java convertește automat operandii utilizând metoda statică `toString()`.

Limbajul Java pune la dispoziție următoarele metode:

- `toString()` - convertește o valoare de un anumit tip în `String`;
- `parseInt()`, `parseLong()`, `valueOf()` - convertește o valoare de tip `String` la `Integer` sau `Long`;
- `floatValue()` - convertește un tip wrapper la un tip predefinit.

Exemplul 4.2.8.

```
// conversie de la int la String:
int n = 123;
String s = Integer.toString(n); //sirul s devine "123"
// conversie de la float la String:
float f = 12.34f;
String s = Float.toString(f); // s devine "12.34"
// conversie de la String la int
String s = "1234";
int n = Integer.parseInt(s); // n devine 1234
// conversie de la String la int
String s = "12.34";
Float temp = Float.valueOf(s); // nu exista parseFloat
float f = temp.floatValue(); // f devine 12.34
```

Prezentăm mai jos câteva metode din clasa `String` (`java.lang.String`):

- `public boolean equals(Object unObiect);`
Compară două șiruri de caractere (obiectul apelat și argumentul propriu-zis).

Exemplul 4.2.9. Se compară șirurile de caractere "Ion" cu "John", care bineînțeles că nu sunt egale:

```
String s="Ion";
if (s.equals("John"))
    System.out.println("Ion este egal cu John");

• public boolean equalsIgnoreCase(String altSir);
Se compară obiectul apelat (convertit la majuscule) și argumentul altSir (convertit la majuscule) și se returnează true dacă sunt egale ca valoare și false în caz contrar.
```

Exemplul 4.2.10. La compararea șirurilor "ion" și "Ion" se returnează true:

```
sir="ion";
if (sir.equalsIgnoreCase("Ion"))
    System.out.println("OK");

• public int compareTo(String altSir);
Returnează zero, dacă șirurile sunt egale, o valoare negativă, dacă șirul obiect precede șirul parametru, iar o valoare pozitivă, dacă șirul obiect urmează șirului parametru în ordine lexicografică.
```

Exemplul 4.2.11. Șirul obiect este "abc", iar cel parametru este "ade". Variabila `n` va primi o valoare negativă.

```
int n = "abc".compareTo("ade"); //n este negativ
if (n == 0)
    rezultat = "sirurile sunt egale";
else
    if (n < 0)
        rezultat = "sir1 < sir2"; // Se va ajunge aici
    else
        rezultat = "sir1 > sir2";

• public String replace(char caracterVechi, char caracterNou);
Înlocuiește în șirul obiect orice apariție a lui caracterVechi cu caracterNou.
```

Exemplul 4.2.12. Se va înlocui litera `m` din șirul "mama" cu litera `t` și astfel va rezulta șirul "tata".

```
sir = "mama".replace('m', 't'); // sir = "tata"

• public String toLowerCase();
Convertește orice majusculă în minusculă a șirului obiect.

• public String trim();
Șterge spațiile albe de la capetele unui șir de caractere, inclusiv \n, \t.
```

Exemplul 4.2.13. Se vor elimina spațiile albe de la începutul și sfârșitul `String`-ului `sir1`, iar rezultatul va fi atribuit lui `sir2`.

```
String sir1=" Un exemplu \n";
```

```
String sir2=sir1.trim(); //sir2 devine "Un exemplu"
• public int length();
  Întoarce numărul de caractere dintr-un șir obiect.
```

Exemplul 4.2.14.

```
int n = "Programarea in Java".length(); // n va fi 19
Reamintim că un tablou obișnuit avea o dată numită length (și nu metodă).
• public String substring(int indexStart, int indexSfarsit);
  Extrage din șir partea specificată (subșirul care începe cu poziția indexStart și se
  termină cu poziția indexSfarsit). Prima poziție a unui șir este 0 și ultimul caracter
  are indexul length()-1 (adică indexSfarsit = indexStart + numărul de
  caractere dintre ele).
```

Exemplul 4.2.15.

```
sir1 = "pozitie";
sir2 = sir1.substring(2, 4); // sir2 = "zi"
• public char charAt(int index);
  Întoarce caracterul de la poziția precizată de variabila index. Numărătoarea
  începe de la zero.
```

Exemplul 4.2.16.

```
c = "pozitie".charAt(5); // c devine 'i'
c = "pozitie".charAt(1); // c devine 'o'
• public int indexOf(String sir, int indexStart);
  Determină indexul de la care se găsește sir în șirul obiect începând de la poziția
  indexStart. Dacă nu este găsit, atunci se returnează -1.
```

Exemplul 4.2.17.

```
int n = "abcdabcd".indexOf("bc",3); // n devine 5
• public boolean endsWidth(String sufix);
  Întoarce true dacă String-ul sufix termină șirul obiect.
```

Exemplul 4.2.18.

```
boolean e = "abcdabcd".endsWith("bcd"); // e va fi true
• public char[] toCharArray()
  Returnează un tablou cu caracterele din șirul obiect. Nu este necesară alocarea de
  spațiu pentru respectivul tablou.
```

Exemplul 4.2.19.

```
String text = "Convertim String in sir de caractere";
char [] sirCaractere = text.toCharArray();
```

```
System.out.println("" + sirCaractere[0] + sirCaractere[10]);
// afiseaza CS
```

```
• public void getChars(int srcBegin, int srcEnd, char[] dst,
  int dstBegin);
  Se copiază caracterele din șirul obiect începând cu poziția srcBegin până la
  poziția srcEnd în tabloul de caractere dst începând cu poziția dstBegin. În acest
  caz, trebuie să alocăm spațiu pentru tabloul dst.
```

Exemplul 4.2.20.

```
String text = "Convertim subString in sir de caractere";
char [] sirCaractere = new char[3];
text.getChars(10, 13, sirCaractere, 0);
System.out.println("" + sirCaractere[0] + sirCaractere[1] +
  sirCaractere[2]);
// se va tipari "sub"
```

```
• public byte[] getBytes(String charsetName)
  throws UnsupportedOperationException
  Caracterele sunt extrase într-un tablou de tip byte. Conversia dintre codul Unicode
  și valorile pe 8 biți (byte) va fi în concordanță cu codificarea sistemului (octetul
  nesemnificativ de la caracterul Unicode va fi convertit în ASCII). În acest caz, nu este
  necesară alocarea spațiului pentru tablou.
```

Exemplul 4.2.21.

```
String text = "Convertim String in sir de caractere";
byte [] sirCaractere = text.getBytes();
• public static String copyValueOf(char[] data);
  Această metodă statică creează un String din vectorul de caractere data.
```

Exemplul 4.2.22. Secvența de cod Java va afișa textul "Un String".

```
char[] textVector = {'U', 'n', ' ', ' ', 'S', 't', 'r', 'i', 'n', 'g'};
String text = String.copyValueOf(textVector);
System.out.println(text);
```

```
• public static String copyValueOf(char[] data, int offset,
  int count);
  Creează un String dintr-o submulțime de elemente ale unui șir. Secvența de
  count caractere din șirul data începând cu poziția offset va forma noul String.
```

Exemplul 4.2.23. Următorul cod Java va conduce la afișarea cuvântului String.

```
char[] textVector = {'U', 'n', ' ', ' ', 'S', 't', 'r', 'i', 'n', 'g'};
String text = String.copyValueOf(textVector, 3, 6);
System.out.println(text); // va fi afisat "String"
```

4.3. Clasa StringBuffer (java.util.StringBuffer)

O altă clasă care se referă la procesarea intensivă a șirurilor de caractere este `StringBuffer`. Ea se ocupă de procesarea șirurilor de caractere de lungime variabilă. O instanță a clasei `StringBuffer` reprezintă un `String` care poate fi modificat dinamic. Am văzut că un obiect `String` nu poate fi modificat, dar am creat obiecte noi de tip `String` obținute prin modificarea obiectelor `String` existente. Șirurile din clasa `StringBuffer` își pot schimba conținutul sau/și lungimea prin alocarea de nou spațiu de memorie.

Un obiect `StringBuffer` conține un bloc de memorie numit `buffer` care poate sau nu să conțină un `String`, și dacă da, nu este obligatoriu să fie ocupat în totalitate. Așadar, lungimea unui șir ținut de un obiect `StringBuffer` poate fi diferită de lungimea `bufferului` (lungimea `bufferului` se numește capacitatea obiectului). Pentru a verifica valoarea capacității unui obiect `StringBuffer` putem apela metoda `capacity()` din clasa `StringBuffer`.

Crearea unui obiect din clasa `StringBuffer` se realizează în mod obișnuit apelând unul dintre constructorii:

- `StringBuffer()`
- `StringBuffer(int capacitate)`
- `StringBuffer(String sir)`

Primul constructor creează un șir cu capacitatea 16 caractere, al doilea cu capacitatea specificată. Ultimul constructor permite inițializarea șirului de caractere, iar capacitatea va fi egală cu lungimea șirului inițial plus 16.

Exemplul 4.3.1. Crearea unui obiect `StringBuffer`:

```
StringBuffer a = new StringBuffer("Un prim exemplu de String dinamic");
```

Exemplul 4.3.2. Aflarea lungimii unui șir și capacitatea `StringBuffer`-ului:

```
StringBuffer a = new StringBuffer("abcdef");
int lungime1 = a.length(); // va fi 6
int capacitate1 = a.capacity(); // va fi 22
StringBuffer b = new StringBuffer();
int lungime2 = b.length(); // va fi 0
int capacitate2 = b.capacity(); // va fi 16
```

Implicit, capacitatea unui obiect `StringBuffer` este lungimea obiectului `StringBuffer` la care se adună 16. Metoda `ensureCapacity()` permite să modificăm valoarea implicită a capacității unui obiect `StringBuffer`. Algoritmul de calcul al noii valori implicite este următorul:

- dacă valoarea implicită a capacității este mai mare decât noua capacitate, atunci capacitatea implicită rămâne aceeași;

- dacă valoarea capacității noi este mai mică decât dublul capacității implicite + 2, atunci capacitatea implicită devine dublul capacității + 2;
- altfel, capacitatea implicită devine egală cu noua capacitate.

Exemplul 4.3.3.

```
StringBuffer a = new StringBuffer("0123456789");
int lungime = a.length(); // va fi 10
int capacitate = a.capacity(); // va fi 26
a.ensureCapacity(30);
lungime = a.length(); // va fi 10
capacitate = a.capacity(); // va fi 54
```

Schimbarea lungimii unui obiect `StringBuffer` se realizează cu ajutorul metodei `setLength()`. Când creștem lungimea unui obiect `StringBuffer`, caracterele în plus vor fi inițializate cu `'\u0000'`. Când descreștem lungimea unui obiect `StringBuffer`, caracterele de la sfârșitul cuvântului se vor trunchia. Metoda `setLength()` nu afectează capacitatea unui obiect `StringBuffer` decât dacă lungimea nouă este mai mare decât capacitatea. În acest caz, noua capacitate va fi dublul vechii capacități + 2. Dacă totuși noua lungime depășește capacitatea, atunci aceasta devine egală cu noua lungime.

Exemplul 4.3.4. Restrângerea lungimii unui șir va conduce la trunchierea acestuia:

```
StringBuffer a = new StringBuffer("0123456789");
a.setLength(8); // a va fi "01234567"
```

Metoda `append()` permite adăugarea unui șir de caractere la sfârșitul unui obiect `StringBuffer` existent. Lungimea șirului conținut de obiectul `StringBuffer` va fi mărită (automat) cu lungimea șirului adăugat. Metoda `append()` returnează obiectul `StringBuffer` extins, deci îl putem asigna unui alt obiect `StringBuffer`.

Exemplul 4.3.5.

```
StringBuffer a = new StringBuffer("abc");
StringBuffer b = a.append("def"); // a și b devin "abcdef"
// de fapt a și b vor pointa asupra aceleiași zone de memorie
b.setLength(4);
System.out.println(a); // a va fi "abcd"
```

O altă versiune de `append()` este cu încă doi parametri care oferă posibilitatea adăugării unui subșir la unul de caractere dat. Acești doi parametri suplimentari sunt poziția indexului primului caracter care trebuie adăugat și numărul total de caractere care trebuie adăugate.

Exemplul 4.3.6. Din `String`-ul `b` se va adăuga subșirul "fg".

```
StringBuffer a = new StringBuffer("abc");
String b = "defghi";
a.append(b, 2, 2); // a va deveni "abcfg"
```

Se pot adăuga la un obiect `StringBuffer` orice variabile din tipurile de bază: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`. În fiecare caz, valoarea este convertită la un `String` echivalent cu valoarea care se adaugă la obiect.

Exemplul 4.3.7.

```
StringBuffer a = new StringBuffer("abc");
long x = 99;
a.append(x); // a devine "abc99"
char [] text = {'d', 'e', 'f', 'g', 'h', 'i'};
a.append(text, 2, 3); // a devine "abc99fgh"
```

Putem de asemenea insera `String`-uri (de fapt valori de variabile din tipurile de bază) în orice poziție a unui obiect `StringBuffer` (nu neapărat la sfârșitul acestuia). În clasa `StringBuffer` există mai multe forme de inserare, printre care:

```
insert(int index, String sir);
insert(int index, char[] str, int pozitie, int lungime);
```

Exemplul 4.3.8. Se va insera șirul "def" în șirul "abc" pe poziția 1.

```
StringBuffer a = new StringBuffer("abc");
a.insert(1, "def"); // a va defini "adefbc"
```

Schimbarea unui singur caracter dintr-un obiect `StringBuffer` se poate realiza cu ajutorul metodei `setCharAt()`. Primul argument indică indexul caracterului care trebuie schimbat, iar al doilea argument, caracterul care trebuie introdus.

Exemplul 4.3.9. Se va înlocui caracterul de pe poziția 1 (b) cu caracterul 'd'.

```
StringBuffer a = new StringBuffer("abc");
a.setCharAt(1, 'd'); // a va defini "adc"
```

Putem obține un obiect `String` dintr-un obiect `StringBuffer` utilizând metoda `toString()` a clasei `StringBuffer`. Această metodă creează un nou obiect `String` care este inițializat cu șirul conținut de obiectul `StringBuffer`.

Exemplul 4.3.9. Variabila b va conține șirul "abc def".

```
StringBuffer a = new StringBuffer("abc def");
String b = a.toString();
```

Metoda `toString()` este intensiv folosită de compilator împreună cu metoda `append()` pentru a implementa concatenarea obiectelor `String`.

Exemplul 4.3.10. Instrucțiunea:

```
String a = "abc" + "def";
```

va fi interpretată de compilator drept:

```
String a = new StringBuffer().append("abc").append("def").
toString();
```

4.4. Clasa StringTokenizer (java.util.StringTokenizer)

Clasa `StringTokenizer` este utilă pentru împărțirea unui șir într-o secvență de atomi lexicali (eng. *token*). Aceștia sunt definiți ca o mulțime de caractere de delimitare. Delimitatorii obișnuiți sunt: spații, taburi, semne de punctuație (, . ;).

Declararea unei variabile din această clasă se face similar cu alte variabile. Există doi constructori frecvent utilizați:

```
public StringTokenizer(String str);
public StringTokenizer(String str, String delims);
```

Primul constructor creează un obiect din clasa `StringTokenizer`, care împarte șirul dat utilizând caracterele de delimitare standard (space, \r, \t, \n).

Al doilea constructor creează un obiect din clasa `StringTokenizer`, care împarte șirul dat folosind caracterele din șirul delimitator.

Exemplul 4.4.1. Instanțierea unui obiect de clasă StringTokenizer:

```
StringTokenizer d = new StringTokenizer("abc de/ c", " /");
```

Metodele cele mai importante din clasa `StringTokenizer` sunt:

- `public String nextToken();`
Întoarce următorul atom lexical.
- `public String nextToken(String delimitator);`
Setează delimitatorii ca fiind caracterele din șirul dat și returnează următorul atom lexical.

Exemplul 4.4.2. Un prim exemplu de divizare a unui șir de caractere:

```
String data = "22 / 06 / 12 : 30";
String zi, luna, ora, min;
StringTokenizer d = new StringTokenizer(data, " /");
zi = d.nextToken(); // zi = "22"
luna = d.nextToken(); // luna = "06"
ora = d.nextToken(); // ora = "12"
min = d.nextToken(":"); // min = "30"
```

- `public boolean hasMoreTokens();`
Returnează `true`, dacă mai există date care pot forma un atom lexical.

Exemplul 4.4.3. Bucla se va executa atât timp cât mai există token-uri.

```
String exemplu = "4,6 ,7, 10, 15 ,20";
String element;
StringTokenizer listaNumere = new StringTokenizer(exemplu, " ,");
while (listaNumere.hasMoreTokens()) {
    element = listaNumere.nextToken();
    ...
}
```

4.5. Structuri dinamice de date

În această secțiune vom prezenta structurile dinamice de listă (generică) simplu și dublu înălțuită (cozi, stive), arbore binar, clasele Vector, Hashtable, Stack, BitSet, Properties și interfața Enumeration.

4.5.1. Liste simplu înălțuite

Pentru implementarea unei liste simplu înălțuite în Java, definim o clasă (numită Element, deocamdată definită pentru valori întregi) pentru nodul care conține datele și o legătură către următorul nod. Pentru construirea unei liste simplu înălțuite, considerăm o clasă ListaInlantuita care utilizează două noduri (primul și ultimul) și o metodă (numită adaugaNod()) ce adaugă un nod nou la listă. De asemenea, clasa ListaInlantuita mai conține și o metodă pentru afișarea listei create.

Exemplul 4.5.1. Implementarea în Java a listei simplu înălțuite:

```
// clasa care contine metoda main()
public class Lista {
    public static void main(String args[]) {
        ListaInlantuita lista = new ListaInlantuita();
        lista.adaugaNod(new Element(7));
        lista.adaugaNod(new Element(3));
        lista.adaugaNod(new Element(0));
        lista.afiseazaLista();
    }
}

// clasa corespunzatoare unui element al listei
class Element {
    int valoare;
    // date de tip int
    Element urmator;
    // legatura catre urmatorul nod
```

```
Element(int valoareNod) {
    valoare = valoareNod;
    urmator = null;
}

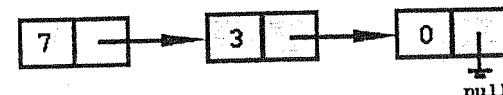
// clasa corespunzatoare listei simplu inlantuite
class ListaInlantuita {
    Element primul, ultimul;
    int numarNoduri;

    ListaInlantuita() {
        primul = ultimul = null;
        numarNoduri = 0;
    }

    // adaugarea se realizeaza la sfarsitul listei
    void adaugaNod(Element nodNou) {
        nodNou.urmator = null;
        if (primul == null) // Acesta este primul element
            primul = nodNou;
        if (ultimul != null)
            ultimul.urmator = nodNou;
        ultimul = nodNou;
        numarNoduri++;
    }

    // afisarea continutului listei
    void afiseazaLista() {
        Element temporar = primul;
        while (temporar != null) {
            System.out.print(temporar.valoare + " ");
            temporar = temporar.urmator;
        }
        System.out.println();
    }
} // de la class ListaInlantuita
```

Lista simplu înălțuită creată de aplicația precedentă poate fi reprezentată grafic astfel:



Continuăm prezentarea listelor simplu înlanțuite cu construirea unei clase generice. Am văzut în exemplul precedent că elementele listei erau toate de același tip (int). Pentru a crea o lista generică, vom stoca referințele la datele din nod, în loc să stocăm datele reale. În exemplul de mai jos, punem în evidență o listă simplu înlanțuită cu elemente de tip int și String. Crearea listei cu ambele tipuri este posibilă trimițând ca argumente pentru nodurile listei obiecte din clasa Integer, respectiv String.

Exemplul 4.5.2. Implementarea unei liste generice.

```
public class ListaGenerica {
    public static void main(String args[]) {
        ListaInlantuitaGenerica listaIntregiStringuri =
            new ListaInlantuitaGenerica();
        listaIntregiStringuri.adaugaNod(new Integer(7));
        listaIntregiStringuri.adaugaNod(new Integer(3));
        listaIntregiStringuri.adaugaNod(new Integer(0));
        listaIntregiStringuri.adaugaNod(new String("sapte"));
        listaIntregiStringuri.adaugaNod(new String("trei"));
        listaIntregiStringuri.adaugaNod(new String("zero"));
        listaIntregiStringuri.afiseazaLista();
    }
}

// clasa corespunzatoare unui element al listei
class ElementGeneric {
    Object valoare;
    // date de tip generic (Object se poate unifica cu un tip
    // arbitrar de date in timpul executiei programului)
    ElementGeneric urmator;
    // legatura catre urmatorul nod

    ElementGeneric(Object valoareNod) {
        valoare = valoareNod;
        urmator = null;
    }
}

// clasa corespunzatoare listei generice simplu inlantuite
class ListaInlantuitaGenerica {
    ElementGeneric primul, ultimul;
    int numarNoduri;

    // constructorul clasei
    ListaInlantuitaGenerica() {
```

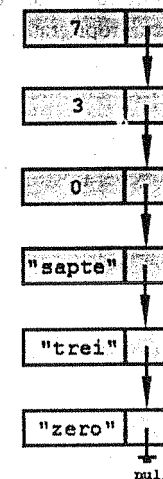
```
        primul = ultimul = null;
        numarNoduri = 0;
    }

    // adaugarea unui nod la lista
    void adaugaNod(Object dateNoi) {
        ElementGeneric nodNou = new ElementGeneric(dateNoi);
        nodNou.urmator = null;
        if (primul == null) // Acesta este primul element
            primul = nodNou;
        if (ultimul != null)
            ultimul.urmator = nodNou;
        ultimul = nodNou;
        numarNoduri++;
    }

    // afisarea continutului listei
    void afiseazaLista() {
        ElementGeneric temporar = primul;
        while (temporar != null) {
            System.out.print(temporar.valoare + " ");
            temporar = temporar.urmator;
        }
        System.out.println();
    }
}

// de la class ListaInlantuitaGenerica
```

Reprezentarea grafică a listei generice simplu înlanțuite din programul anterior este:



Din categoria listelor înlănțuite, există două cazuri particulare des utilizate: coada și stiva. *Coadă* este o listă de elemente care adaugă elemente noi la sfârșitul listei și elimină elementele de la începutul listei. Această structură de date este cunoscută și sub denumirea de FIFO (First In, First Out). Structura de tip coadă este des folosită în programele de parcurgere sistematică a grafurilor, închidere tranzitivă a unei relații, traversarea în lățime a arborilor, probleme de punct fix etc.

Stiva este o listă de elemente care adaugă și șterge elemente de la sfârșitul listei. Această structură de date este cunoscută și sub denumirea de LIFO (Last In, First Out). Structura de stivă este des folosită în programe de parcurgere sistematică a grafurilor, traversarea în adâncime a arborilor, evaluarea expresiilor aritmetice etc.

4.5.2. Arbori binari

Se numește *arbore* un graf (orientat) finit, conex și fără circuite. Un arbore este *binar* dacă orice nod din arbore are cel mult doi descendenți (fii) direcți. Se cunosc trei tipuri de parcurgeri standard ale arborilor, în funcție de ordinea de „vizitare” a rădăcinii, subarborelui stâng și a subarborelui drept:

- *preordine* (rădăcină, fiu stâng, fiu drept);
- *inordine* (fiu stâng, rădăcină, fiu drept);
- *postordine* (fiu stâng, fiu drept, rădăcină).

În practică, sunt foarte des utilizați arborii de căutare și ansamblele cu proprietatea de maxim (minim). Un *arbore binar* este numit *de căutare* dacă, pentru orice nod N, eticheta descendentului stâng este mai mică decât eticheta nodului N, iar eticheta descendentului drept este mai mare decât eticheta nodului N. Arborii binari de căutare sunt utilizați în probleme de căutare eficientă a unui element. Parcurgerea în inordine a unui arbore binar de căutare furnizează ordinea crescătoare a elementelor.

Un arbore binar este *ansamblu* cu proprietatea de maxim, dacă pentru orice nod N, etichetele descendenților direcți sunt mai mici decât eticheta nodului N. Ansamblele cu proprietatea de maxim sunt folosite la metoda (optimă) de sortare HeapSort.

Exemplul 4.5.3. Se citesc cuvinte din fișierul `numere.txt` care vor fi inserate într-un arbore de căutare. Se va realiza parcurgerea acestuia în preordine, inordine și, respectiv, postordine.

```
import java.io.*;
import java.util.*;

// definim clasa NodArboreBinar folosita pentru reprezentarea
// unui arbore binar de cautare
class NodArboreBinar {
    String cuvant;
    int contorCuvant;
```

```
NodArboreBinar stanga, dreapta;
```

```
// constructorul clasei
NodArboreBinar (String cuvantNou) {
    cuvant = cuvantNou;
    contorCuvant = 1;
    stanga = dreapta = null;
}
}
```

```
// definim clasa OperatiiArboreBinar care cuprinde descrierea
// operatiilor principale din arbore
```

```
class OperatiiArboreBinar {
    NodArboreBinar radacina;
```

```
// constructorul clasei
OperatiiArboreBinar() {
    radacina = null;
}
```

```
void insereazaNod(NodArboreBinar nod, String cuvantNou) {
    int compara = nod.cuvant.compareTo(cuvantNou);
    if (compara == 0) // cuvantul exista deja
        nod.contorCuvant ++;
    else
        if (compara > 0) // ne ducem la stanga
            if (nod.stanga != null)
                insereazaNod(nod.stanga, cuvantNou);
            else
                nod.stanga = new NodArboreBinar(cuvantNou);
        else // compara < 0
            if (nod.dreapta != null)
                insereazaNod(nod.dreapta, cuvantNou);
            else
                nod.dreapta = new NodArboreBinar(cuvantNou);
}
```

```
// aceasta metoda are menirea de a separa cazul crearii
// radacinii de cazul inserarii unui nod ordinar
void insereazaCuvant(String cuvantNou) {
    if (radacina == null)
        radacina = new NodArboreBinar(cuvantNou);
    else
```



```

        insereazaNod(radacina, cuvantNou);
    }

    // metoda cauta nodul care contine cuvantul specificat
    // parametrul nod indica arborele in care se realizeaza
    // cautarea
    NodArboreBinar gaseste(NodArboreBinar nod, String
    cuvantCheie) {
        int compara = nod.cuvant.compareTo(cuvantCheie);
        if (compara == 0) // am gasit cuvantCheie
            return nod;
        else
            if (compara > 0)
                if (nod.stanga != null)
                    return gaseste(nod.stanga, cuvantCheie);
                else
                    return null; // nu a fost gasit cuvantCheie
            else // compara < 0
                if (nod.dreapta != null)
                    return gaseste(nod.dreapta, cuvantCheie);
                else
                    return null;
    }

    int cuvantContor(String cuvantCheie) {
        if (radacina != null) {
            NodArboreBinar nod = gaseste(radacina, cuvantCheie);
            if (nod != null)
                return nod.contorCuvant;
            else
                return 0;
        }
        return 0; // arbore vid
    }

    void preordine(NodArboreBinar nod, StringBuffer
    listaPreordine) {
        // in lista preordine, am preferat sa listam
        // si nodurile vide in ideea refacerii
        // ulterioare a structurii de arbore binar.
        // Mai precis, intre reprezentarea
        // arborelui si lista in preordine
        // completa, exista o corespondenta biunivoca.

```

```

        if (nod != null) {
            String pereche = "(" + nod.cuvant + ", " + nod.contorCuvant
            + ") \n";
            listaPreordine.append(pereche);
            if (nod.stanga != null)
                preordine(nod.stanga, listaPreordine);
            else
                preordine(nod.stanga, listaPreordine.append("(null, 0) "));
            if (nod.dreapta != null)
                preordine(nod.dreapta, listaPreordine);
            else
                preordine(nod.dreapta, listaPreordine.append(
                "(null, 0) "));
        }
    }

    void inordine(NodArboreBinar nod, StringBuffer
    listaInordine) {
        if (nod != null) {
            String pereche = "(" + nod.cuvant + ", " +
            nod.contorCuvant + ") \n";
            inordine(nod.stanga, listaInordine);
            listaInordine.append(pereche);
            inordine(nod.dreapta, listaInordine);
        }
    }

    void postordine(NodArboreBinar nod, StringBuffer
    listaPostordine) {
        if (nod != null) {
            String pereche = "(" + nod.cuvant + ", " +
            nod.contorCuvant + ") \n";
            postordine(nod.stanga, listaPostordine);
            postordine(nod.dreapta, listaPostordine);
            listaPostordine.append(pereche);
        }
    }
} // sfarsitul clasei OperatiiArboreBinar

public class cuvantArboreBinar {
    public static void main(String args[]) {
        OperatiiArboreBinar arboreBinar = new
        OperatiiArboreBinar();
    }
}

```

```

BufferedReader intrare;
try {
    intrare = new BufferedReader(
        new FileReader("numere.txt"));
    String oLinie;
    while ((oLinie = intrare.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(oLinie,
            " ,;\t\n\r");
        while (st.hasMoreTokens()) {
            String sir = st.nextToken();
            arboreBinar.insereazaCuvant(new String(sir));
        }
    }
    intrare.close();
}
catch (Exception e) {
    System.out.println("Eroare: " + e);
}
// probam pentru cateva cuvinte
System.out.println("Numar cuvinte 'JAVA' = " +
    arboreBinar.cuvantContor("JAVA"));
System.out.println("Numar cuvinte 'AVAJ' = " +
    arboreBinar.cuvantContor("AVAJ"));
StringBuffer listaPreordine = new StringBuffer("");
arboreBinar.preordine(arboreBinar.radacina, listaPreordine);
System.out.println("Lista in preordine (completa) este " +
    listaPreordine);
StringBuffer listaInordine = new StringBuffer("");
arboreBinar.inordine(arboreBinar.radacina, listaInordine);
System.out.println(
    "Lista in inordine (ce corespunde cu sortarea" +
    " crescatoare) este " + listaInordine);
StringBuffer listaPostordine = new StringBuffer("");
arboreBinar.postordine(arboreBinar.radacina, listaPostordine);
System.out.println("Lista in postordine este " +
    listaPostordine);
}
}

```

Dacă fișierul numere.txt conține următoarele două linii:

```

IS 20 ALF
JAVA AVAJJ

```

atunci execuția programului va duce la următorul afișaj pe ecran:

```

Numar cuvinte 'JAVA' = 1
Numar cuvinte 'AVAJ' = 0

```

Lista in preordine (completa) este (IS, 1)

```

(20, 1)
(null, 0) (ALF, 1)
(null, 0) (AVAJJ, 1)
(null, 0) (null, 0) (JAVA, 1)
(null, 0) (null, 0)

```

Lista in inordine (ce corespunde cu sortarea crescatoare) este

```

(20, 1)
(ALF, 1)
(AVAJJ, 1)
(IS, 1)
(JAVA, 1)

```

Lista in postordine este (AVAJJ, 1)

```

(ALF, 1)
(20, 1)
(JAVA, 1)
(IS, 1)

```

4.5.3. Enumeratori

Toate colecțiile pot fi văzute ca secvențe liniare de elemente. Accesul la un element al unei colecții diferă de la o colecție la alta. De exemplu, un vector este indexat după o cheie număr întreg pozitiv, iar o tabelă hash poate utiliza orice tip de obiect drept cheie.

Pentru accesul unitar la elementele unei colecții, se folosește interfața Enumeration care specifică două metode:

- `hasMoreElements()` - returnează o valoare booleană care indică dacă mai sunt elemente de enumerat;
- `nextElement()` returnează următorul element din enumerare.

Aceste operații sunt folosite pentru formarea unei instrucțiuni repetitive. Metodele `hasMoreElements()` și `nextElement()` trebuie utilizate în tandem. Adică se apelează întâi `hasMoreElements()` și abia apoi `nextElement()`. De asemenea, nu se apelează `nextElement()` a doua oară fără să se fi apelat în prealabil `hasMoreElements()`.

Exemplul 4.5.4. Parcurgerea unei colecții se poate realiza astfel:

```

for (Enumeration e = htab.elements(); e.hasMoreElements(); ) {
    System.out.println(e.nextElement());
}

```

Cu excepția șirurilor, toate colecțiile Java furnizează metode care returnează o enumerare. În plus, și alte clase care nu sunt neapărat colecții suportă protocolul de enumerare. De exemplu, clasa `StringTokenizer` este utilizată pentru extragerea de

cuvinte dintr-o valoare `String`. Cuvintele individuale sunt apoi accesate folosind metode de enumerare.

4.5.3.1. Colecția Vector

Tipul de dată `Vector` este similar cu șirurile de obiecte. Cu toate acestea, el furnizează un număr mare de operații de nivel înalt care nu sunt suportate de șiruri. Mai important, un vector este expandabil, adică dimensiunea lui crește, dacă sunt inserate noi elemente la vector. Pentru utilizarea acestui tip de dată trebuie importată clasa `java.util.Vector`. Un obiect de tip `Vector` se poate crea folosind operatorul `new`. Deoarece obiectele de tip `Vector` trebuie să fie din subclase ale lui `Object`, înseamnă că un vector nu poate stoca valori primitive, cum ar fi întregii sau realii. Cu toate acestea, clasele *wrapper* pot fi utilizate pentru a converti asemenea valori la obiecte (și reciproc).

Cele mai frecvent utilizate operații furnizate de clasa `Vector` sunt:

a) Determinarea dimensiunii:

- `size()` - returnează numărul de elemente din colecție ca valoare de tip `int`;
- `isEmpty()` - returnează `true` dacă colecția este vidă;
- `capacity()` - returnează capacitatea curentă a unui vector ca un `int`;
- `setSize(int lungimeNoua)` - setează lungimea unui vector, trunchiind sau adăugând, dacă este necesar.

b) Accesul la elemente:

- `contains(Object element)` - determină dacă o valoare este sau nu în vector;
- `firstElement()` - returnează primul element al colecției;
- `lastElement()` - returnează ultimul element al colecției;
- `elementAt(int index)` - returnează elementul memorat la o poziție dată.

c) Inserție și modificare:

- `addElement(Object valoare)` - adaugă noua valoare la sfârșitul colecției;
- `setElementAt(Object valoare, int index)` - setează la indexul precizat valoarea dată;
- `insertElementAt(Object valoare, int index)` - inserează valoarea la indexul precizat.

d) Ștergere:

- `removeElementAt(int index)` - șterge elementul de la indexul precizat, reducând dimensiunea vectorului;
- `removeElement(Object valoare)` - șterge toate instanțele egale cu valoare;
- `removeAllElements()` - șterge toate valorile din vector.

e) Căutare:

- `indexOf(Object valoare)` - returnează indexul primei apariții;
- `lastIndexOf(Object valoare)` - returnează indexul ultimei apariții.

f) Diverse:

- `clone()` - returnează o copie a vectorului;
- `toString()` - returnează reprezentarea ca șir a vectorului.

Clasa `Vector` poate simula șirurile, listele (stivele, cozile), mulțimile etc. În cele ce urmează, vom prezenta două exemple în care folosim structura `Vector`.

Exemplul 4.5.5. Se creează un vector cu informații despre studenți.

```
import java.util.Vector;

class Student {
    int numarMatricol;
    String nume, prenume;
    float media;

    Student(int numarMatricol, String nume, String prenume, float media) {
        this.numarMatricol = numarMatricol;
        this.nume = new String(nume);
        this.prenume = new String(prenume);
        this.media = media;
    }

    public String toString() {
        return("NumarMatricol:" + numarMatricol + " Nume: " + nume +
            " Prenume: " + prenume + " Media: " + media);
    }
} // sfarsitul clasei Student

public class VectorStudenti {
    public static void main(String args[]) {
        Vector listaStudenti = new Vector();
        listaStudenti.addElement(new Student(52321, "Trufanda",
            "Viorel", 9.99f));
        listaStudenti.addElement(new Student(52322, "Raicu",
            "Florin", 8.99f));
        listaStudenti.addElement(new Student(52426, "Munteanu",
            "Loredana", 9.49f));
        for (int i = 0; i < listaStudenti.size(); i++) {
            Student curent = (Student) listaStudenti.elementAt(i);
            System.out.print("Studenta " + i + ": ");
            System.out.println(curent);
        }
    }
} // sfarsitul clasei VectorStudenti
```

Exemplul 4.5.6. Conversia dintr-un șir de întregi (`int`) la un `Vector` de `Integer`.

```
import java.util.*;
```

```

public class UtilizareVectori {
    public static void main(String args[]) {
        int tablouFix [] = {1, 3, 0, 5, -3, 0, 3, 4, 0, 2, 0, 0,
            5, 12345};
        Vector listaIntregi = new Vector();
        // variabila listaIntregi va contine elementele din
        // tablouFix[] care sunt nenule
        int i; // variabila de lucru
        int elementCurent;
        // se parcurge sirul de intregi
        for (i = 0; i < tablouFix.length; i++) {
            elementCurent = tablouFix[i];
            if (elementCurent != 0)
                // daca elementul este nenul
                // se adauga in lista de intregi
                listaIntregi.addElement(new Integer(elementCurent));
        }
        // afisarea listei de intregi
        for (i = 0; i < listaIntregi.size(); i++) {
            // se obtine elementul de pe pozitia i
            int valoare = ((Integer) listaIntregi.elementAt(i)).
                intValue();
            System.out.println("Element " + i + ": " + valoare);
        }
    }
} // sfarsitul clasei FolosireVectori

```

Ca și listele generice, vectorii au posibilitatea de a memora obiecte provenind din clase diferite. Acest lucru este posibil, deoarece obiectele Vector utilizează o referință Object pentru a indica spre elementele lor. Astfel, se poate utiliza un obiect Vector pentru a stoca orice tip de obiecte în cadrul acelui obiect Vector. Singura cerință este de a avea posibilitatea de a determina ce tipuri de obiecte stochează vectorul și cum le organizează. În acest fel, programul poate converti elementul la tipul corect.

Limbajul Java pune la dispoziție metoda public final Class getClass(), care returnează clasa obiectului în momentul execuției. Această metodă se poate folosi împreună cu metoda public String getName() care returnează numele complet al tipului (clasei, interfeței, șirului sau primitivei), reprezentată de obiectul de tip Class, ca un String.

Exemplul 4.5.7. Evidențiază modul de utilizare a vectorilor generici.

```

import java.util.Vector;

class ClasaGenerica {

```

```

    String nume;

    ClasaGenerica(String nume) {
        this.nume = new String(nume);
    }

    public String toString() {
        return(" Nume: " + nume);
    }
} // sfarsitul clasei ClasaGenerica

public class VectorGeneric {
    public static void main(String args[]) {
        Vector listaGenerica = new Vector();
        listaGenerica.addElement(new ClasaGenerica("Claudia"));
        listaGenerica.addElement(new Integer(52322));
        listaGenerica.addElement(new String("abc"));
        for (int i = 0; i < listaGenerica.size(); i++) {
            Object obiectCurent = listaGenerica.elementAt(i);
            if (obiectCurent.getClass().getName().equals(
                "ClasaGenerica")) {
                System.out.println("Obiect din clasa ClasaGenerica:"
                    + obiectCurent);
            }
            if (obiectCurent.getClass().getName().equals(
                "java.lang.Integer")) {
                System.out.println("Obiect din clasa Integer:"
                    + obiectCurent);
            }
            if (obiectCurent.getClass().getName().equals(
                "java.lang.String")) {
                System.out.println("Obiect din clasa String:"
                    + obiectCurent);
            }
        }
    }
} // sfarsitul clasei VectorGeneric

```

Execuția programului va avea ca efect:

```

Obiect din clasa ClasaGenerica: Nume: Claudia
Obiect din clasa Integer: 52322
Obiect din clasa String: abc

```

4.5.4. Tabele de dispersie (hash)

O *tabelă de dispersie* (eng. *hash*) este o structură de date care permite căutarea elementelor stocate utilizând chei asociate. Un tablou permite accesul rapid la un element prin specificarea unui index întreg. Pe de altă parte, o tabelă hash permite asocierea unui element cu o cheie și apoi utilizarea acelei chei pentru a căuta elementul. Pentru a face distincție între un articol sau altul, cheia asociată utilizată trebuie să fie unică pentru fiecare element.

Pentru a construi o tabelă hash în Java, trebuie creat un obiect `Hashtable`. Adăugarea de elemente noi la tabela de dispersie se realizează cu metoda `put()`, al cărei format este:

- `Object put(Object cheie, Object valoare);`
Preluarea unui articol utilizând cheia se realizează cu metoda `get()`, care are următorul prototip:
- `Object get(Object cheie);`
La fel ca în cazul clasei `Vector`, metoda `get()` returnează un `Object` pe care trebuie să-l convertim la clasa inițială a elementului. Eliminarea unui element dintr-o tabelă hash se realizează apelând metoda `remove()`:
- `Object remove(Object cheie);`

Exemplul 4.5.8. Un prim exemplu de utilizare a clasei `Hashtable`:

```
import java.util.*;

class Student {
    String nume, prenume;
    float media;

    Student(String nume, String prenume, float media) {
        this.nume = new String(nume);
        this.prenume = new String(prenume);
        this.media = media;
    }

    public String toString() {
        return("Nume: " + nume + " Prenume: " + prenume + " Media:"
            + media);
    }
} // sfarsitul clasei Student

public class TabelaHashStudenti {
    public static void main(String args[]) {
        Hashtable listaStudenti = new Hashtable();
```

```
        listaStudenti.put("top-1", new Student("Neagu", "Valentin",
            8.80f));
        listaStudenti.put("top-2", new Student("Gabureanu", "Petru",
            9.25f));
        listaStudenti.put("top-7", new Student("Orzan",
            "Alexandrina", 9.67f));
        // preluam un student folosind cheia tabelii Hash
        Student student = (Student) listaStudenti.get("top-7");
        System.out.println("Studentul cu cheia top-7 este: " +
            student);
    }
} // sfarsitul clasei TabelaHashStudenti
```

O tabelă hash își mărește automat dimensiunea atunci când are nevoie de mai multă capacitate de stocare. Ca și în cazul constructorului `Vector`, se poate controla capacitatea unei tabeli hash și modul său de creștere. Astfel, există încă doi constructori în clasa `Hashtable` care setează de la început capacitatea inițială a unei tabeli hash, precum și factorul de creștere:

- `Hashtable(int capacitateInitiala);`
- `Hashtable(int capacitateInitiala, float factor);`

Spre deosebire de clasa `Vector`, obiectele `Hashtable` se redimensionează de obicei înainte să-și utilizeze capacitatea maximă. Implicit, o tabelă hash își dublează dimensiunea atunci când ajunge la acea fracțiune de capacitate specificată prin parametrul `factor`. Parametrul `factor` este un număr în virgulă mobilă cuprins între 0.0 și 1.0. Parametrul `factor` implicit este 0.75, deci atunci când se ajunge la 75% din tabela hash ocupată, dimensiunea sa se dublează. Clasa care implementează tabela hash calculează numărul întreg pentru fiecare cheie și valoarea acesteia trebuie să fie unică și inferioară capacității tabelii. Când o tabelă hash se apropie de limita capacității sale, algoritmul de căutare a numărului index devine mai puțin eficient și performanța tabelii hash scade.

Accesarea tuturor elementelor unei tabeli hash se poate realiza prin intermediul metodei `Elements()` din clasa `Hashtable` care returnează un obiect `Enumeration`.

Exemplul 4.5.9. Considerăm programul `TabelaHashStudenti.java` de mai sus la care adăugăm:

```
Enumeration enum = listaStudenti.elements();
while (enum.hasMoreElements()) {
    student = (Student) enum.nextElement();
    System.out.println(student);
}
```

O metodă importantă pentru lucrul cu tabele hash este public `int hashCode()` care returnează valoarea codului hash al unui obiect. De fiecare dată când este apelată

pentru același obiect în timpul execuției unui program, metoda `hashCode()` returnează același întreg. Mai mult, dacă două obiecte sunt egale conform metodei `equals()`, atunci apelul metodei `hashCode()` aplicat acestor două obiecte furnizează același întreg. Acest întreg nu rămâne însă același de la o execuție la alta a programului.

Exemplul 4.5.10. Considerăm programul de la Exemplul 4.5.8., la care adăugăm:

```
Student student1 = (Student) listaStudenti.get("top-7");
System.out.println("HashCode = " + student1.hashCode());
Student student2 = (Student) listaStudenti.get("top-7");
System.out.println("HashCode = " + student2.hashCode());
Student student3 = (Student) listaStudenti.get("top-1");
System.out.println("HashCode = " + student3.hashCode());
```

4.5.5. Clasa Stack (java.util.Stack)

Clasa `Stack` este derivată din clasa `Vector`. În plus, conține cinci operații care permit unui vector să fie tratat ca o stivă (LIFO – *Last Input First Input*). Există un singur constructor, care este cel implicit, și cinci metode:

- `boolean empty()` – testează dacă stiva este vidă;
- `Object peek()` – întoarce obiectul din vârful stivei fără să-l ștergă;
- `Object pop()` – întoarce obiectul din vârful stivei și-l șterge din stivă;
- `Object push(Object item)` – înserează un obiect în stivă;
- `int search(Object o)` – returnează prima poziție unde este găsit obiectul în stivă. Întregul returnat reprezintă poziția relativă față de vârful stivei, aceasta fiind -1 dacă elementul nu este găsit.

Deoarece clasa `Stack` este derivată din `Vector`, putem utiliza metoda `elements()` din clasa `Vector` care întoarce o enumerare. Ca și la `Vector`, enumerarea va fi creată în ordinea inserării elementelor, și nu de la vârf la bază!

Exemplul 4.5.11.

```
import java.util.*;

public class testStiva {
    public static void main(String args[]) {
        Stack stiva = new Stack();
        stiva.push(new String("unu"));
        stiva.push(new Integer(2));
        stiva.push(new Double(3.0));
        Double d = (Double) stiva.peek();
        System.out.println("elementul din top este " + d);
        Enumeration e = stiva.elements();
        while (e.hasMoreElements()) {
```

```
        Object elementCurent = e.nextElement();
        System.out.print(elementCurent + " ");
    }
} // sfarsitul clasei testStiva
```

4.5.6. Clasa BitSet

Clasa `BitSet` se derivează direct din `Object` pentru a implementa structura de mulțime folosind un vector caracteristic (boolean). Astfel, fiecare componentă are o valoare booleană (true, false). Biții pot fi examinați, setați sau șterși. Biții dintr-un `BitSet` sunt indexați după întregi pozitivi. Un `BitSet` poate fi utilizat pentru a schimba conținutul unui alt `BitSet` folosind operatorii logici AND, OR inclusiv, și OR exclusiv. Implicit, toți biții dintr-un set au valoarea false. Fiecare `BitSet` are o dimensiune curentă, care coincide cu numărul de biți din `BitSet`.

Există doi constructori:

- `public BitSet();`
- `public BitSet(int numarBiți);`

A doua formă permite specificarea dimensiunii inițiale a obiectului `BitSet`. Pentru a specifica dacă un bit din cadrul grupului este setat sau nu, se pot utiliza metodele:

- `public void set(int indexBit);`
- `public void clear(int indexBit);`

Parametrul `indexBit` specifică bitul din obiect care trebuie activat sau nu. Pentru a afla dacă un bit este sau nu setat, se poate utiliza metoda:

- `public boolean get(int indexBit);`

Aceasta returnează true, dacă bitul de la poziția `indexBit` este setat, și false, dacă este șters. Există patru metode care simulează operațiile logice:

- `public void and(BitSet set)` – execută un AND logic dintre obiectul curent și argumentul set;
- `public void andNot(BitSet set)` – șterge toți biții din obiectul curent care sunt setați în set;
- `public void or(BitSet set)` execută un OR logic dintre obiectul curent și argumentul set;
- `public void xor(BitSet set)` execută un XOR logic (sau exclusiv) dintre obiectul curent și argumentul set.

Exemplul 4.5.12. Programul ilustrează modalitatea de utilizare a clasei `BitSet`:

```
import java.util.*;
```

```

public class BitSetTest {
    public static void main(String args[]) {
        BitSet bs1 = new BitSet(4);
        BitSet bs2 = new BitSet(4);
        // setam bs1 = (true, false, true, true)
        bs1.set(0);
        bs1.clear(1);
        bs1.set(2);
        bs1.set(3);
        System.out.println(bs1); // se va afisa {0, 2, 3}
        // setam bs1 cu toti bitii activati
        for (int i = 0; i < 4; i++)
            bs2.set(i);
        // inverseaza bs1 cu bs2, folosind metoda XOR
        bs1.xor(bs2);
        System.out.println(bs1); // se va afisa {1}
    }
} // sfarsitul clasei BitSetTest

```

4.5.7. Clasa Properties

Clasa Properties este subclasă a clasei Hashtable și este utilizată de sistemul de execuție a programelor Java pentru lucrul cu un tip special de tabel hash, așa-numita listă de proprietăți. Clasa Properties lucrează cu o colecție de perechi de șiruri cheie/valoare. Acestea reprezintă valori care descriu mediul de execuție curent (numele utilizatorului, numele sistemului de operare, directorul rădăcină etc.). Listarea proprietăților sistemului se poate realiza prin apelul funcției `getProperties()` din clasa `System`:

- `public static Properties getProperties();`

Exemplul 4.5.13. Programul de mai jos va lista toate proprietățile:

```

import java.util.*;

public class Proprietati {
    public static void main(String args[]) {
        System.getProperties().list(System.out);
    }
}

```

Atunci când programele operează cu obiecte `Hashtable`, va trebui uneori să stocăm și să preluăm conținutul unei tabele dintr-un fișier. O listă de proprietăți permite stocarea de valori implicite. Dacă un program nu poate găsi o anumită proprietate într-o listă, lista poate încerca valoarea implicită din fișierul de proprietăți.

Limitarea unei liste de proprietăți este aceea că atât cheia, cât și valoarea stocată în tabelă trebuie să fie obiecte `String`.

Aplicațiile posibile ale listei de proprietăți sunt cele de personalizare și salvare a configurației între execuțiile unui program. Fișierul listă de proprietăți este „echivalent” cu un fișier `.ini` utilizat sub Windows. Sintaxa este asemănătoare.

Exemplul 4.5.14. Iată un fișier listă de proprietăți (`propApl.ini`):

```

#lista de proprietati - propApl.ini
Foreground=0000FF
Background=FF8888
FontSize=12
Font=Courier

```

Crearea unei liste de proprietăți se face apelând unul dintre constructorii clasei `Properties`:

- `public Properties()` - creează o listă de proprietăți vidă fără valori implicite;
- `public Properties(Properties implicit)` - creează o listă de proprietăți vidă cu valorile inițiale date de implicit.

Adăugarea de proprietăți la listă se realizează cu metoda `put()` (suprascrisă peste cea din `Hashtable`):

- `public Object put(Object cheie, Object valoare)` - inserează valoarea specificată la cheia indicată în mulțimea de proprietăți. Parametrii cheie și valoare nu pot fi `null`. Metoda returnează valoarea precedentă a cheii specificate din `Hashtable` sau `null` dacă nu exista nici una. Metoda poate arunca o excepție `NullPointerException` în cazul în care cheie sau valoare sunt `null`.

O proprietate dintr-o listă se poate obține apelând una din variantele metodei `getProperty()`:

- `public String getProperty(String cheie)` - caută proprietatea cu cheia specificată în lista de proprietăți. Dacă nu este găsită, se caută în lista de proprietăți implicite. Metoda returnează `null` dacă nu este găsită nici o proprietate.
- `public String getProperty(String cheie, String valoare Implicita)` - caută proprietatea cu cheia specificată în lista de proprietăți. Dacă nu este găsită în lista de proprietăți, atunci se caută în lista de proprietăți implicite. Metoda returnează `valoareImplicita` dacă nu este găsită nici o proprietate.

Ca și la obiecte `Hashtable`, putem enumera obiecte `Properties` utilizând metoda:

- `public Enumeration propertyNames()` - returnează o enumerare a tuturor cheilor din lista de proprietăți, inclusiv cu cheile din lista de proprietăți implicite.

Exemplul 4.5.15. Programul reprezintă un model de utilizare a listelor de proprietăți:

```
import java.util.*;

public class ListaProprietati {
    public static void main(String args[]) {
        Properties prop = new Properties();
        prop.put("unu", "1");
        prop.put("doi", "2");
        prop.put("trei", "3");
        prop.put("patru", "4");
        prop.put("cinci", "5");
        Enumeration enum = prop.propertyNames();
        while (enum.hasMoreElements()) {
            String cheieProp = (String) enum.nextElement();
            System.out.println(cheieProp + ": " +
                prop.getProperty(cheieProp));
        }
    }
}
```

Rularea acestui program va avea drept consecință afișarea pe ecran:

```
doi: 2
patru: 4
cinci: 5
unu: 1
trei: 3
```

Ordinea listei enumerate nu este garantată a fi aceeași cu cea de adăugare a elementelor în lista de proprietăți (cum se întâmplă în cazul Hashtable).

Procesul de citire a fișierelor de proprietăți se poate realiza deschizând un flux de intrare către fișierul de proprietăți. Acest lucru este realizat de metoda:

```
public void load(InputStream fluxIntrare) throws IOException
```

care citește o listă de proprietăți (perechi cheie = element) de la fluxul de intrare. Fiecare proprietate ocupă o linie din flux care este terminată cu \n, \r sau \r\n. Liniile din fluxIntrare sunt procesate până când se întâlnește sfârșitul fișierului.

Exemplul 4.5.16. Citirea proprietăților dintr-un fișier:

```
Properties prop = new Properties();
try {
    FileInputStream fluxIntrare = new FileInputStream(
        "fisProp.ini");
    prop.load(fluxIntrare);
}
```

```
}
catch (IOException e) {
    System.out.println("Eroare citire fisier:" + e);
}
```

După inițializare, o listă de proprietăți se poate salva într-un fișier apelând metoda:

```
public void store(OutputStream fluxIesire, String antet) throws
IOException
```

care scrie lista de proprietăți în fluxIesire într-un format adecvat pentru citirea sa într-o listă Properties. Proprietățile din tabela implicită (dacă există) nu sunt scrise în fluxIesire. Dacă argumentul antet este diferit de null, atunci se va scrie ca prima linie caracterul #, șirul antet și o linie separatoare. Astfel, antetul poate servi drept comentariu de identificare.

Exemplul 4.5.17. Salvarea proprietăților într-un fișier:

```
Properties prop = new Properties();

try {
    FileOutputStream fluxIesire = new FileOutputStream(
        "fisProp.ini");
    prop.store(fluxIesire, "Lista studentilor admisi");
}
catch (IOException e) {
    System.out.println("Eroare scriere fisier: " + e);
}
```

Exemplul 4.5.18. Următoarea aplicație permite utilizatorului schimbarea culorii de fundal, a culorii textului, a tipului și dimensiunii fontului pentru o aplicație, utilizând un fișier listă de proprietăți, numit fisProp.ini. Acesta poate fi:

```
#lista de proprietati - fisProp.ini
```

```
Foreground=0000FF
Background=FF8888
FontSize=16
Font=Courier
```

Fișierul Personalizare.java este:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
```

```
public class Personalizare extends Frame implements
WindowListener {
```



```

TextArea zonaText = new TextArea(
    "Schimbam culorile si fontul");
Font f;

public static void main(String[] args) {
    Personalizare aplicatie = new Personalizare(
        "PersonalizareAplicatie");
    aplicatie.setSize(300, 400);
    aplicatie.show();
}

public Personalizare(String nume) {
    super(nume);
    add("Center", zonaText);
    Properties prop = new Properties();
    try {
        FileInputStream fluxIntrare = new FileInputStream(
            "fisProp.ini");
        prop.load(fluxIntrare);
    }
    catch (IOException e) {
        System.out.println("Eroare citire fisier:" + e);
    }
    // setam culoarea Background-ului
    String arg = prop.getProperty("Background");
    int valoareCuloare = Integer.valueOf(arg, 16).intValue();
    zonaText.setBackground(new Color(valoareCuloare));
    // setam culoarea Foreground-ului
    arg = prop.getProperty("Foreground");
    valoareCuloare = Integer.valueOf(arg, 16).intValue();
    zonaText.setForeground(new Color(valoareCuloare));
    // setam stilul fontului
    String stilFont = prop.getProperty("Font");
    arg = prop.getProperty("FontSize");
    int dimFont = Integer.valueOf(arg).intValue();
    f = new Font(stilFont, Font.PLAIN, dimFont);
    zonaText.setFont(f);
    addWindowListener(this);
}

// definirea metodelor din interfata WindowListener
public void windowClosing(WindowEvent event) {
    System.exit(0);
}

```

```

public void windowIconified(WindowEvent event) {};
public void windowDeiconified(WindowEvent event) {};
public void windowOpened(WindowEvent event) {};
public void windowClosed(WindowEvent event) {};
public void windowActivated(WindowEvent event) {};
public void windowDeactivated(WindowEvent event) {};
} // sfarsitul clasei Personalizare

```

4.6. Concluzii

Capitolul de față a debutat cu prezentarea a trei clase de bază pentru crearea, afișarea și manipularea șirurilor de caractere: `String`, `StringBuffer` și `StringTokenizer`. Se observă faptul că operațiile cu șiruri de caractere se pot realiza foarte ușor utilizând clasele prezentate.

Structurile dinamice sunt frecvent utilizate în practică pentru ușurința și eficiența acestora. S-a prezentat lucrul cu liste înlănțuite, arbori, enumeratori, tabele de dispersie, stive, mulțimi și proprietăți.

4.7. Test grilă

Întrebarea 4.7.1. Un obiect de tip `String` permite parcurgerea caracter cu caracter a literalului stocat?

- Există o metodă care se ocupă cu lucrul acesta.
- Se poate realiza cu ajutorul mai multor metode din clasa `String`.
- Se poate realiza doar prin intermediul clasei `StringBuffer`.
- Această operație poate fi efectuată și cu ajutorul unui obiect de tip `StringTokenizer`.

Întrebarea 4.7.2. Ce se poate spune despre următoarea secvență de cod?

```

String s = "String";
s.toString().toString().toString();

```

- Se obține eroare la compilare, deoarece nu s-a utilizat operatorul `new` pentru obiectul `s`.
- Apare o eroare la compilare, întrucât obiectul `s` va conține numele unui cuvânt rezervat.
- Se va genera o eroare la execuție, pentru că metoda `toString()` returnează o valoare care nu este preluată.
- Codul este corect din punct de vedere sintactic.
- Se apelează metode `toString()` pentru același obiect.

Întrebarea 4.7.3. Care dintre metodele clasei `String` de mai jos returnează un obiect de tip `String`?

- `replace()`

- b) `split()`
- c) `intern()`
- d) `toLowerCase()`

Întrebarea 4.7.4. Ce va afișa la consolă următorul apel?

```
System.out.println(10 + 9 + new StringBuffer("opt") + 7 + 6 + "5")
```

- a) 109opt765
- b) 19opt135
- c) 19opt18
- d) 19opt765
- e) Se obține o eroare la compilare sau la execuție.

Întrebarea 4.7.5. Pentru a gestiona numele unor persoane, care structură este mai eficientă pentru căutare?

- a) o colecție de tip vector;
- b) o tabelă hash;
- c) o listă dublu înlănțuită;
- d) un arbore binar;
- e) o mulțime.

4.8. Exerciții propuse spre implementare

Exercițiul 4.8.1. Completați clasa `ListaInlantuita` (Exemplul 4.5.1.) implementând metode pentru inserare, eliminare, căutare, numărare de elemente ale unei liste simplu înlănțuite.

Exercițiul 4.8.2. Scrieți un program Java care tratează cazul listelor dublu înlănțuite implementând toate operațiile specifice listelor simplu înlănțuite.

Exercițiul 4.8.3. Scrieți un program Java care implementează operațiile de inserare (*push*), extragere (*pop*), afișare, căutare pentru structurile de coadă și stivă.

Exercițiul 4.8.4. Scrieți un program Java care, utilizând ansamblu cu proprietatea de maxim, sortează un șir de elemente folosind metoda `HeapSort`.

Exercițiul 4.8.5. Utilizând clasa `Stack`, scrieți un program Java care simulează un calculator de buzunar. Implementați operațiile `+`, `-` (unar și binar), `*`, `/` și radical. Testați condiția de pozitivitate la radical (returnați un mesaj propriu de eroare), cazul împărțirii la 0 sau când se obține un număr care nu este în domeniu.

Exercițiul 4.8.6. Scrieți o clasă proprie care extinde clasa `Stack` și care implementează afișarea, numărarea, oglindirea elementelor unei stive. Atenție! Stiva este generică,

deci precizați și din ce clasă face parte elementul referit (puteți folosi metodele `getClass()` și `getName()`).

Exercițiul 4.8.7. Folosind clasa `BitSet`, scrieți un program Java care execută reuniunea, intersecția a două mulțimi, precum și complementara a două mulțimi. Mulțimile se citesc din două fișiere și au elemente din mulțimea $\{0, \dots, k\}$, k fixat strict pozitiv (de exemplu, $k = 1000$).

Exercițiul 4.8.8. Definiți o clasă Java care implementează interfața `Enumeration` menită să enumere structura de arbore binar (clasa trebuie să definească cele două metode din această interfață: `hasMoreElements()` și `nextElement()`). Puteți utiliza o stivă pentru liniarizarea arborelui folosind parcurgerea lui în preordine.

5. Fișiere, fluxuri de date și serializarea obiectelor

5.1. Cuvinte cheie

- fișiere, fluxuri de intrare/ieșire, buffere, filtre
- serializare, deserializare
- procesarea fișierelor

5.2. Introducere

Fișierele de intrare și ieșire Java implică utilizarea *fluxurilor* (eng. *streams*). Un *flux* este o reprezentare abstractă a unui dispozitiv de intrare sau ieșire care este sursă, respectiv destinație pentru date. Astfel, putem scrie date într-un flux sau citi date dintr-un flux. De fapt, putem vedea fluxul ca o secvență de octeți care „curge” (eng. *flows*) într-un program Java.

Când se scriu date într-un flux, fluxul se numește *flux de ieșire*. Fluxul de ieșire poate circula către orice dispozitiv unde se poate transfera o secvență de octeți, cum ar fi un fișier de pe un hard disc local, un fișier de pe un sistem de la distanță sau o imprimantă. Se poate direcționa un flux de ieșire către ecran, dar acesta va afișa doar informații textuale, nu și grafice, care necesită un suport specializat (vom reveni în capitolul de ferestre grafice).

Putem citi date dintr-un *flux de intrare*. Aceasta poate fi orice sursă serială de date care este de obicei un fișier pe hard discul local, tastatură sau de pe un calculator de la distanță.

Fără a intra în detalii de sisteme de operare, în principiu există două posibilități de memorare a datelor (informațiilor) în calculatorul local:

- a) *memoria internă* (eng. RAM – Random Access Memory);
- b) *memoria externă*, cum ar fi dispozitivele de memorare a fișierelor (floppy-disk, hard-disk, CD, DVD etc.).

De obicei, fluxurile de intrare și ieșire ale mașinii pe care se execută aplicația sunt disponibile în aplicații Java, nu și appleturilor. În caz contrar, un applet apelat dintr-o pagină Web ar putea accesa hard discul local creând eventuale probleme de securitate. Încercarea de a accesa fișiere ale discului local dintr-un applet Java este prohibită de obicei prin aruncarea unei excepții `IOException`. Cu toate acestea, directorul în care se află appletul și subdirectoarele acestuia sunt accesibile pentru applet. În plus, se pot folosi proprietățile de securitate Java pentru controlul unui applet (și a unei aplicații care rulează sub un manager de securitate) putând accesa fișiere și alte resurse pentru care există permisiune explicită.

Principalul motiv al utilizării unui flux ca bază pentru operațiile de intrare/ieșire în Java este independența codului față de dispozitivul implicat. Avantajele ar fi *transparența*, nu trebuie să știm detaliile fizice ale fiecărui dispozitiv, și *portabilitatea*, programul va funcționa pe diferite dispozitive fără a schimba sursa sau codul obiect Java.

Metodele fluxurilor de intrare/ieșire permit citirea/scrierea unei cantități mici de date (un octet, un caracter etc.), deci transferul datelor în acest mod către un dispozitiv fizic (de exemplu, un hard disc) ar fi extrem de inefficient. Pentru rezolvarea acestei probleme, un flux are atașat un buffer (eng. *buffered stream*). Un buffer este un bloc de memorie folosit pentru a reține date din/către un dispozitiv extern.

Fluxurile cu buffer asigură că transferul datelor între memorie și un dispozitiv extern folosesc un buffer suficient de mare pentru ca operațiile de intrare/ieșire să fie

eficiente. Când se scrie într-un flux de ieșire bufferizat, datele sunt trimise către buffer și nu direct către dispozitivul extern. Cantitatea de date din buffer este automat transferată și când acesta s-a umplut, datele sunt transferate către dispozitivul extern. Uneori se dorește ca datele să fie trimise către dispozitiv înaintea umplerii complete a bufferului. Există metode care eliberează conținutul bufferului (înainte ca acesta să fie plin) și trimit datele către dispozitivul extern. Această operație se numește golirea bufferului (eng. *flushing the buffer*). Fluxurile de intrare bufferizate lucrează la fel, adică orice operație de citire dintr-un flux de intrare bufferizat va citi date din buffer, iar când acesta va fi gol, se vor transfera date din dispozitivul extern către buffer (dacă sunt suficiente date).

Pachetul `java.io` oferă două tipuri de fluxuri: *binare* (care conțin date binare) și *caracter* (care conțin date caracter). În cazul fluxurilor binare, scrierea/citirea datelor într-un flux se face ca o serie de octeți, exact cum acestea apar în memorie. Nu este nevoie de vreo transformare a datelor. Java utilizează scrierea internă a caracterelor Unicode, așadar când se scriu caractere către un flux ca date binare, fiecare caracter pe 16 biți este reprezentat pe doi octeți, cel semnificativ fiind scris primul.

Fluxurile caracter sunt folosite pentru memorarea și obținerea textului sau pentru a citi un fișier text care nu a fost scris de un program Java. Toate datele numerice sunt convertite unei reprezentări text înainte de a fi scrise în flux. Aceasta implică formatarea datelor pentru a genera reprezentarea caracter a valorilor datelor. Citirea datelor numerice dintr-un flux care conține text implică mai multă muncă decât citirea datelor binare. Citirea dintr-un flux presupune că se cunoaște câte caractere formează o valoare numerică și convertește corect caracterele din flux. Trebuie să se recunoască începutul și sfârșitul fiecărei valori numerice și apoi se realizează conversia atomului lexical (eng. *token*), adică a secvenței de caractere care reprezintă acea valoare, în formă binară a valorii.

Exemplul 5.2.1. Presupunem că avem declarațiile:

```
int i = 123, j = 45;
```

Valorile acestor variabile sunt reprezentate în memorie (pe opt octeți) ca: `0x7B`, respectiv `0x2D`. În momentul trimiterii către un flux, se va converti fiecare cifră a valorii întregi în caracter ASCII, obținându-se secvența de cinci octeți: `0x31`, `0x32`, `0x33`, respectiv `0x34`, `0x35`. Pentru conversia inversă, va trebui să știm că primii trei octeți se referă la prima valoare întreagă (formată din cifrele 1, 2, 3) și următorii doi octeți la cea de-a doua valoare (formată din cifrele 4, 5).

Când se scriu caractere într-un flux ca date caracter, caracterele Unicode sunt convertite automat către reprezentarea locală a caracterelor (folosite de mașina gazdă) și apoi acestea sunt scrise în fișier. Când se citește un șir de caractere, datele din fișier sunt convertite automat din reprezentarea mașinii locale către caractere Unicode. În cazul fluxurilor caracter, programul citește și scrie caractere Unicode, dar fișierul va conține caractere din codificarea caracterelor echivalentă utilizată de calculatorul local. Fluxul suport este întotdeauna o secvență de octeți, indiferent dacă se scriu/citește date binare sau caractere.

Java utilizează două feluri de reprezentări text și codificare a caracterelor:

- Unicode – pentru reprezentarea internă a caracterelor și a String-urilor;
- UTF – pentru intrări și ieșiri.

Unicode utilizează 16 biți pentru reprezentarea unui caracter. În cazul în care cei mai semnificativi nouă biți sunt toți 0, atunci codificarea este simplă standard ASCII (ultimii 7 biți fiind reprezentarea caracterelor). Altfel, biții reprezintă un caracter care nu este reprezentat în ASCII pe 7 biți. Tipul `char` din Java folosește codificarea Unicode (de asemenea clasa `String`).

Codificarea Unicode este suficientă pentru majoritatea limbilor, dar cele asiatice grafice reprezintă o problemă (sunt prea multe caractere grafice). Soluția constă în utilizarea UTF (Universal Character Set Transformation Format). Codificarea UTF folosește exact numărul dorit de biți: mai puțini pentru alfabet mic, mai mulți pentru alfabet mare.

O codificare a caracterelor este o corespondență bijectivă dintre mulțimea caracterelor și o mulțime de numere binare. Fiecare platformă Java are o codificare a caracterelor implicită, care este folosită pentru interpretarea dintre Unicode intern și octeți externi. Codificarea implicită a caracterelor reflectă limba și cultura locală. Fiecare codificare are un nume. De exemplu, 8859-1 înseamnă ASCII, 8859-8 este ISO Latin/Hebrew și CP1258 este vietnameză.

Exemplul 5.2.2. Următorul subprogram Java va returna numele codificării caracterelor de pe platforma Java (de exemplu CP1250).

```
try {
    FileInputStream fis = new FileInputStream("fisier.in");
    InputStreamReader isr = new InputStreamReader(fis);
    System.out.println("Codificarea este " + isr.getEncoding());
} catch (IOException e) {
    System.out.println("Eroare scriere fisier " + e);
}
```

Când are loc o operație de intrare/ieșire în Java, sistemul trebuie să știe care este codificarea caracterelor. Clasele I/O utilizează de obicei codificarea implicită locală, în caz că nu se precizează una explicită. Cu toate că este adecvată codificarea implicită locală, în aplicațiile de rețea, când se comunică cu un alt computer, acestea trebuie să folosească aceeași codificare. În acest caz, este recomandată cererea explicită 8859-1.

5.3. Clase Java pentru intrări și ieșiri

Toate clasele de intrare/ieșire se găsesc în pachetul `java.io`. Vom începe prin prezentarea claselor `File` și `RandomAccessFile` care furnizează funcționalitate pentru navigare pe sistemul local, descriind fișierele, directoarele și accesul direct (nesecvențial) la fișiere. Accesul secvențial al fișierelor se realizează cu *fluxuri de citire* (eng. *readers*) și *fluxuri de scriere* (eng. *writers*) pe care le vom prezenta ulterior.

5.3.1. Clasa File

Clasa `File` pune la dispoziție facilități pentru manipularea fișierelor și directoarelor. Programele Java care folosesc această clasă trebuie să conțină instrucțiunea de import:

```
import java.io.*;
```

Clasa `java.io.File` reprezintă numele unui fișier sau director care poate exista pe sistemul de fișiere a mașinii gazdă. O cale în structura de directoare (sub Windows) se afișează folosind separatorul „\” (eng. *backslash*), cum ar fi de exemplu `c:\j2sdk1.4.1\bin\javac.exe`. Pe un sistem de operare UNIX, calea se scrie cu „/” (eng. *slash*), cum ar fi `/home/stefan/java/testGrila.java`.

Acest separator se poate determina folosind constanta (din clasa `File`):

```
public final static String separator;
```

Calea este absolută în exemplele precedente, deoarece este specificată de la directorul rădăcină până la fișier. Dacă de exemplu suntem în directorul `stefan`, atunci calea relativă este `java/testGrila.java`. Cea mai simplă formă de constructor pentru aceasta clasă este

```
File(String numeCale);
```

Construcția unei instanțe a lui `File` nu creează un fișier pe sistemul local, ci doar o instanță care încapsulează șirul specificat. De exemplu, iată cum putem declara și crea o instanță a clasei `File`:

```
File fis = new File("c:\j2sdk1.4.1\bin\javac.exe");
```

Reamintim că „\” înseamnă „\”, deoarece „\” este un caracter cu semnificație specială (eng. *escape*) și separator de fișiere în sistemul de operare Windows.

Mai există două variante de constructori din clasa `File`:

a) `File(String dir, String subCale);`

De exemplu, furnizăm un director și o cale absolută, astfel:

```
File f = new File("/tmp", "xyz");
```

b) `File(File dir, String subCale);` de exemplu, furnizăm o instanță a lui `File` și o cale relativă, astfel:

```
File f1 = new File("C:\\tmp");
```

```
File f2 = new File(f1, "xyz.java");
```

Iată câteva metode utile pentru lucrul cu fișiere:

- `public String getPath();`
Furnizează calea relativă; exemplu:
`String caleRelativa = fis.getPath();`
- `public String getAbsolutePath();`
Furnizează calea absolută; exemplu:
`String caleAbsoluta = fis.getAbsolutePath();`
- `public String getCanonicalPath();`

Furnizează calea canonică a unui fișier sau director. Este similară cu metoda `getAbsolutePath()`, dar sunt rezolvate (identificate) simbolurile `.` (directorul curent) și `..` (directorul părinte).

- `public boolean exists();`
Returnează `true`, dacă fișierul sau directorul specificat există.
- `public boolean isDirectory();`
Întoarce `true`, dacă fișierul există și este director.
- `public boolean isFile();`
Returnează `true`, dacă fișierul există pe sistemul de fișiere și este un fișier ordinar.
- `public long length();`
Returnează lungimea fișierului în octeți sau 0, dacă nu există; exemplu:
`long lungime = fis.length();`
- `public String[] list();`
Returnează un șir de nume de fișiere din directorul specificat de obiectul `File`; exemplu: `String [] continutDirector = fis.list();` Dacă fișierul `fis` nu este director, atunci funcția va întoarce `null`;
- `boolean canRead();`
Returnează `true`, dacă fișierul sau directorul poate fi citit.
- `boolean canWrite();`
Returnează `true`, dacă fișierul sau directorul poate fi modificat.
- `boolean delete();`
Returnează `true`, dacă fișierul sau directorul a fost șters.
- `boolean mkdir();`
Returnează `true`, dacă s-a reușit crearea unui director a cărui cale este descrisă de obiectul `File`.
- `boolean renameTo(File noulNume);`
Returnează `true`, dacă s-a putut redenumi fișierul sau directorul, altfel returnează `false`.

Exemplul 5.3.1. Următorul program pune în evidență crearea (în caz că nu există deja) a directorului `C:\dirProba` (sub sistemul Windows):

```
import java.io.*;

public class Fisiere {
    public static void main(String args[]) {
```

```
        int i;
        boolean reusita;
        File dirNou = new File("C:\\dirProba");
        if (dirNou.isDirectory())
            System.out.println(
                "Directorul C:\\dirProba exista deja!");
        reusita = dirNou.mkdir();
        if (reusita)
            System.out.println("Am reusit sa cream directorul " +
                "C:\\dirProba");
        else
            System.out.println("Nu am creat directorul " +
                "C:\\dirProba!");
    }
}
```

5.3.2. Clasa `RandomAccessFile`

Această clasă se ocupă de lucrul cu fișiere binare (cu acces aleator direct), și nu cu fluxurile fișierelor de caractere (cu acces secvențial). Într-un fișier cu acces direct, putem căuta o dată de la o anumită poziție, putem scrie sau citi date la o poziție precizată, pe când într-un fișier cu acces secvențial putem citi datele în ordinea în care apar. Clasa `java.io.RandomAccessFile` are doi constructori:

```
RandomAccessFile(String fisier, String mod);
RandomAccessFile(File fisier, String mod);
```

String-ul `mod` poate fi `"r"`, dacă se deschide fișierul în acces de citire, sau `"rw"`, dacă se deschide în acces de citire și scriere. A doua formă de constructor este utilă atunci când există deja o instanță a clasei `File`.

Exemplu 5.3.2. Înainte de a deschide un fișier în acces de citire/scriere, facem niște verificări (dacă este fișier – și nu director, dacă se poate citi și dacă se poate scrie în el):

```
File fisier = new File(cale);
if (!fisier.isFile() || !fisier.canRead() || !
    fisier.canWrite()) {
    throw new IOException();
}
RandomAccessFile raf = new RandomAccessFile(fisier, "rw");
```

Metodele cele mai importante din această clasă sunt:

- `long getFilePointer()` throws `IOException`;
Returnează poziția curentă a pointerului din fișier (în octeți) (returnează valoarea deplasamentului).

- `long length()` throws `IOException`;
Returnează lungimea fișierului, în octeți.
- `void seek(long pozitie)` throws `IOException`;
Setează poziția curentă din fișier (în octeți). Fișierele încep cu poziția 0.

Metodele care suportă citirea și scrierea de octeți sunt:

- `int read()` throws `IOException`;
Returnează următorul octet din fișier (memorat în octetul cel mai nesemnificativ al unei variabile de tip `int`) sau valoarea -1, dacă pointerul este la sfârșitul fișierului.
- `int read(byte destinatie[])` throws `IOException`;
Încearcă să citească suficienți octeți pentru a umple șirul `destinatie[]`. Returnează numărul de octeți citați din fișier sau valoarea -1, dacă pointerul este la sfârșitul fișierului.
- `int read(byte destinatie[], int deplasament, int lungime)` throws `IOException`;
Încearcă să citească lungime octeți în șirul `destinatie[]`, începând de la deplasament. Returnează numărul de octeți citați din fișier sau valoarea -1, dacă pointerul este la sfârșitul fișierului.
- `int write(int b)` throws `IOException`;
Scrie octetul cel mai nesemnificativ al lui `b`.
- `int write(byte b[])` throws `IOException`;
Scrie toți octeții din șirul `b[]`.
- `int write(byte destinatie[], int deplasament, int lungime)` throws `IOException`;
Scrie lungime octeți din șirul `destinatie[]`, începând de la poziția deplasament.

Clasa `RandomAccessFile` are metode pentru citirea și scrierea tuturor tipurilor primitive de date. Aceste metode sunt:

- `boolean readBoolean();`
- `void writeBoolean(boolean b);`
- `byte readByte();`
- `void writeByte(int b);`
- `short readShort();`
- `void writeShort(int s);`
- `char readChar();`
- `void writeChar(int c);`
- `int readInt();`
- `void writeInt(int i);`
- `long readLong();`
- `void writeLong(long l);`
- `float readFloat();`

- `void writeFloat(float f);`
- `double readDouble();`
- `void writeDouble(double d);`
- `int readUnsignedByte();`
- `int readUnsignedShort();`
- `String readLine();`
- `String readUTF();`
- `void writeUTF(String s);`

Când un fișier cu acces aleator nu mai este folosit, atunci acesta trebuie închis prin apelul metodei:

- `void close()` throws `IOException`;
eliberându-se astfel resursele sistem de memorie asociate fișierului.

Exemplul 5.3.3. Fie programul Java de mai jos:

```
import java.io.*;

public class FisiereAccesAleator {
    public static void main(String args[]) {
        try {
            RandomAccessFile fisierUnu = new RandomAccessFile(
                "fisier.bin", "rw");
            fisierUnu.writeInt(10000);
            fisierUnu.writeInt(20000);
            fisierUnu.writeInt(30000);
            // vom inlocui in fisier valoarea 20000 cu 40000
            fisierUnu.seek(4);
            fisierUnu.writeInt(40000);
            fisierUnu.close();
        }
        catch (IOException e) {
            System.out.println("Eroare de scriere in fisierUnu " +
                e.toString());
        }
        try {
            // fisier.bin va contine valorile intregi 10000,
            // 40000 si 30000
            RandomAccessFile fisierDoi = new RandomAccessFile(
                "fisier.bin", "r");
            // citim primul intreg
            int i = fisierDoi.readInt();
            System.out.println("i1 = " + i + " pointer = " +
                fisierDoi.getFilePointer());
        }
    }
}
```

```

// citim al doilea intreg
i = fisierDoi.readInt();
System.out.println("i2 = " + i + " pointer = " +
    fisierDoi.getFilePointer());
// resetam pointerul catre fisierDoi la zero
fisierDoi.seek(0);
// citim patru octeti care formeaza un int
byte b = fisierDoi.readByte();
// afisam octetul cel mai semnificativ
System.out.println("b4 = " + b);
b = fisierDoi.readByte();
System.out.println("b3 = " + b);
b = fisierDoi.readByte();
System.out.println("b2 = " + b);
b = fisierDoi.readByte();
// afisam octetul cel mai nesemnificativ
System.out.println("b1 = " + b);
// citim urmatorii opt octeti si-l interpretam ca double
double d = fisierDoi.readDouble();
System.out.println("d = " + d);
fisierDoi.close();
}
catch (IOException e) {
    System.out.println("Eroare de citire din fisier " +
        e.toString());
}
}

```

Mai întâi se va crea fișierul binar (prin acces de scriere) `fișier.bin`, în care sunt trei întregi: 10000, 20000 și 30000. Apoi poziționăm pointerul la fișier după patru octeți (prin apelarea metodei `seek()`) și astfel vom înlocui 20000 cu 40000. Cu apelul metodei `getFilePointer()` vom verifica faptul că o variabilă de tip `int` se memorează într-un fișier binar pe patru octeți. Apoi, poziționăm pointerul la fișier pe prima poziție și citim pe rând primii patru octeți (utilizând o variabilă de tip `byte`) și apoi ultimii opt octeți folosind o variabilă de tip `double`. Execuția programului Java de mai sus va afișa:

```

i1 = 10000 pointer = 4
i2 = 40000 pointer = 8
b4 = 0
b3 = 0
b2 = 39
b1 = 16
d = 8.4879831653433E-310

```

5.3.3. Clase Java pentru fluxuri de intrare/ieșire

Clasele Java pentru fluxuri de intrare/ieșire consideră intrarea și ieșirea ca secvențe ordonate de octeți. Uneori datele apar ca octeți, întregi, numere reale etc. Clasele Java pentru fluxuri de intrare/ieșire pun la dispoziție o abordare structurată asemănătoare celei din clasa `RandomAccessFile`. Acestea sunt:

- fluxuri de ieșire de nivel jos care primesc și scriu octeți la un dispozitiv de ieșire;
- fluxuri de ieșire filtru de nivel înalt care primesc date în format general (primitive) și scriu octeți către un flux de ieșire de nivel jos sau către alt flux de ieșire filtru;
- fluxuri de scriere (eng. *writers*) sunt asemănătoare cu fluxurile de ieșire filtru specializate pe scriere de șiruri Java în unități de caractere Unicode;
- fluxuri de intrare de nivel jos care citesc octeți de la un dispozitiv de intrare și returnează octeții apelantului;
- fluxuri de intrare filtru de nivel înalt care citesc date dintr-un flux de intrare de nivel jos sau de la un alt flux de intrare filtru și returnează date în format general (primitive) către apelant;
- fluxurile de citire (eng. *readers*) sunt asemănătoare cu fluxurile de intrare filtru specializate pe citire de șiruri UTF în unități de caractere Unicode.

Pachetul JDK 1.0.2 pune la dispoziție pentru lucrul cu fluxuri clasele `InputStream/OutputStream`, iar începând cu JDK 1.1, clasele `Reader/Writer`. Primele clase lucrează cu fluxuri de octeți (unitatea de transfer fiind de 8 biți), în timp ce ultimele lucrează cu fluxuri de caractere, care în Java sunt reprezentate pe 16 biți în format Unicode. Avantajul fluxurilor pe caractere Unicode este că oferă un suport conceptului de internaționalizare. Acest concept permite dezvoltarea de appleturi Java care să se adapteze setărilor calculatorului pe care rulează, setări legate de limbă, zona geografică, ora locală etc.

Clasele `Reader/Writer` sunt fluxuri de caractere care nu sunt dependente de tipul de codare a caracterelor. Cu toate că noile clase `Reader/Writer` dublează clasele `InputStream/OutputStream`, acestea din urmă nu au fost abandonate (pentru păstrarea compatibilității cu aplicațiile mai vechi). Mai mult, s-au adăugat clasele de compresie la familia claselor vechi. Totuși, există metode din vechile clase care se recomandă să nu fie folosite, fiind învechite (eng. *deprecated*). Cele mai importante clase noi de intrare/ieșire sunt: `Reader`, `BufferedReader`, `InputStreamReader`, `FileReader`, `Writer`, `PrintWriter` și `FileWriter`.

În general, se obișnuiește pentru fișiere utilizarea claselor `BufferedReader` și `PrintWriter` (citiri și scrieri de linii de text), iar pentru intrări și ieșiri care nu sunt fișiere (de exemplu: tastatură, pagini Web etc.) se folosește clasa `InputStreamReader`. Programele Java care utilizează fișiere trebuie să importe pachetul `java.io.*`. Cum spuneam și în Secțiunea 5.2., *buffer* înseamnă o zonă tampon între dispozitivele de memorare a fișierelor (care sunt mai lente) și memoria RAM (la care există viteză ridicată de acces).

5.3.3.1. Fluxuri de nivel jos

Fluxurile de intrare de nivel jos au metode care citesc și returnează intrarea ca octeți. Fluxurile de ieșire de nivel înalt au metode care trimit octeți și scriu octeți la ieșire. Vom începe discuția prin prezentarea claselor `FileInputStream` și `FileOutputStream`.

Cei mai utilizați constructori pentru fluxurile fișiere de intrare sunt:

- `FileInputStream(String cale);`
- `FileInputStream(File fisier);`

După ce un flux fișier de intrare a fost construit, putem apela metode de citire a unui octet, șir de octeți, porțiuni de șiruri de octeți (asemănător cu cele din clasa `RandomAccessFile`). Acestea sunt:

- `int read() throws IOException;`
Returnează următorul octet din fișier (memorat în octetul cel mai ne semnificativ al unei variabile de tip `int`) sau `-1`, dacă pointerul este la sfârșitul fișierului.
- `int read(byte destinatie[]) throws IOException;`
Încearcă să citească suficienți octeți pentru a umple șirul `destinatie[]`. Returnează numărul de octeți citați din fișier sau `-1`, dacă pointerul este la sfârșitul fișierului.
- `int read(byte destinatie[], int deplasament, int lungime) throws IOException;`
Încearcă să citească lungime octeți în șirul `destinatie[]`, începând de la valoarea lui `deplasament`. Returnează numărul de octeți citați din fișier sau `-1`, dacă pointerul este la sfârșitul fișierului.

Clasa `FileInputStream` conține metodele utile:

- `int available() throws IOException;`
Returnează numărul de octeți care pot fi citați fără blocare.
- `void close() throws IOException;`
Eliberează resursele sistem de memorie asociate fișierului. Un flux fișier de intrare trebuie închis mereu când nu mai este nevoie de el.
- `long skip(long numOcteti) throws IOException;`
Încearcă să citească și să se deplaseze cu `numOcteti` octeți. Returnează numărul de octeți cu care s-a deplasat în fișier.

Fluxurile fișiere de ieșire sunt aproape identice cu cele de intrare. Cei mai utilizați constructori pentru fluxurile fișiere de ieșire sunt:

- `FileOutputStream(String cale);`
- `FileOutputStream(File fisier);`

Există metode pentru scrierea unui octet, șir de octeți sau unui subșir de octeți:

- `void write(int b) throws IOException;`

Scrie octetul cel mai ne semnificativ al lui `b`.

- `void write(byte b[]) throws IOException;`
Scrie toți octeții din șirul `b[]`.
- `void write(byte destinatie[], int deplasament, int lungime) throws IOException;`
Scrie lungime octeți din șirul `destinatie[]`, începând de la poziția `deplasament`.

Clasa `FileOutputStream` oferă metoda `close()`, care trebuie apelată neapărat când nu mai este nevoie de fluxul fișier de ieșire.

Exemplul 5.3.4. Următorul program Java testează lucrul cu fluxuri fișiere `FileOutputStream`, respectiv `FileInputStream`:

```
import java.io.*;

public class FluxuriFișiere {
    public static void main(String args[]) {
        int i;
        try {
            FileOutputStream fisierUnu = new FileOutputStream(
                "fisier.txt");
            // declarăm date de tip byte
            byte b = 5;
            // tipul byte = {-128, ..., 0, ..., 127}, deci necesită
            // un octet pentru memorare
            byte vectorUnu [] = {(byte)2, (byte)-128, (byte)127,
                                (byte)0, (byte)6, (byte)-8};
            fisierUnu.write(b);
            fisierUnu.write(vectorUnu);
            fisierUnu.close();
            FileInputStream fisierDoi = new FileInputStream(
                "fisier.txt");
            int bInt = fisierDoi.read();
            // funcția read() întoarce -1 dacă am ajuns la sfârșitul
            // fișierului
            System.out.println(bInt + " = bInt");
            System.out.println("disponibil = " +
                fisierDoi.available());
            b = (byte) fisierDoi.read();
            System.out.println(b + " = b");
            b = (byte) fisierDoi.read();
            System.out.println(b + " = b");
            byte vectorDoi[] = new byte[10];
            int spatiulRamas = fisierDoi.available();
```

```

System.out.println("disponibil = " +
    fisierDoi.available());
fisierDoi.read(vectorDoi);
for (i = 0; i < spatiulRamas; i++)
    System.out.println("vectorDoi[" + i + "] = " +
        vectorDoi[i]);
b = (byte) fisierDoi.read();
// functia read() intoarce -1 daca am ajuns la
// sfarsitul fisierului
System.out.println(b + " ");
fisierDoi.close();
}
catch (IOException e) {
    System.out.println("Eroare scriere fisier " + e);
}
}

```

Programul va afișa la execuție:

```

5 = bInt
disponibil = 6
2 = b
-128 = b
disponibil = 4
vectorDoi[0] = 127
vectorDoi[1] = 0
vectorDoi[2] = 6
vectorDoi[3] = -8
-1

```

Pachetul `java.io` mai conține și alte clase despre fluxuri fișiere de intrare și ieșire de nivel jos, printre care:

- `InputStream` și `OutputStream`. Acestea sunt superclase pentru toate clasele de fluxuri de nivel jos. De exemplu, se pot utiliza pentru citirea și scrierea socketurilor de rețea;
- `ByteArrayInputStream` și `ByteArrayOutputStream`. Aceste clase citesc și scriu șiruri de octeți. Șirurile de octeți nu sunt dispozitive de intrare/ieșire, dar clasele sunt utile pentru procesarea și crearea secvențelor de octeți;
- `PipedInputStream` și `PipedOutputStream`. Aceste clase furnizează un mecanism pentru comunicare sincronizată dintre fire de execuție.

5.3.3.2. Fluxuri filtru de nivel înalt

Uneori este bine să citim din fluxuri informații de nivel înalt, cum ar fi întregi, `String`-uri etc. Java oferă suport pentru fluxuri de nivel înalt care se extind din superclasele `FilterInputStream` și `FilterOutputStream`. Fluxurile de intrare de nivel înalt nu pot citi direct din dispozitive de intrare, cum ar fi fișiere sau socket-uri. Ele citesc din alte fluxuri. Fluxurile de ieșire de nivel înalt nu pot scrie în dispozitive de ieșire, dar pot scrie în alte fluxuri.

Un exemplu de clasă des utilizată este `DataInputStream`, care are un singur constructor:

- `DataInputStream(InputStream fluxIntrare);`

Argumentul constructorului poate fi un flux fișier de intrare (`FileInputStream` extinde clasa `InputStream`), un flux de intrare dintr-un socket sau din alte tipuri de fluxuri de intrare. Când este apelată instanța unui obiect `DataInputStream`, aceasta va realiza un număr de apeluri `read()` către `fluxIntrare`, să proceseze octeții și să returneze o valoare convenabilă. Cele mai utilizate metode ale clasei `DataInputStream` sunt:

- `boolean readBoolean()` throws `IOException`;
- `byte readByte()` throws `IOException`;
- `short readShort()` throws `IOException`;
- `char readChar()` throws `IOException`;
- `int readInt()` throws `IOException`;
- `long readLong()` throws `IOException`;
- `float readFloat()` throws `IOException`;
- `double readDouble()` throws `IOException`;
- `String readUTF()` throws `IOException`;
- `void close()` throws `IOException`;

Clasa `DataOutputStream` este asemănătoare cu clasa `DataInputStream` și pune la dispoziție constructorul:

- `DataOutputStream(OutputStream fluxIesire);`

Constructorul necesită primirea unui flux de ieșire. Când scriem într-un flux de ieșire, acesta convertește parametrii metodei de scriere în octeți și îi scrie în flux `Iesire`. Cele mai utilizate metode ale clasei `DataOutputStream` sunt:

- `void writeBoolean(boolean b)` throws `IOException`;
- `void writeByte(int b)` throws `IOException`;
- `void writeBytes(String s)` throws `IOException`;
- `void writeShort(int s)` throws `IOException`;
- `void writeChar(int c)` throws `IOException`;
- `void writeInt(int i)` throws `IOException`;
- `void writeLong(long l)` throws `IOException`;
- `void writeFloat(float f)` throws `IOException`;

- void writeDouble(double d) throws IOException;
- void writeUTF(String s) throws IOException;
- void close() throws IOException;

Exemplul 5.3.5. Următorul program Java testează lucrul cu fluxuri filtru Data OutputStream, respectiv DataInputStream:

```
import java.io.*;

public class FluxuriFiltru {
    public static void main(String args[]) {
        try {
            FileOutputStream fisierUnu = new FileOutputStream(
                "fisierUnu.out");
            DataOutputStream fisierDateUnu = new DataOutputStream(
                fisierUnu);
            fisierDateUnu.writeByte(127);
            fisierDateUnu.writeInt(123456);
            fisierDateUnu.writeDouble(12.34);
            fisierDateUnu.writeUTF("123\u5555");
            fisierDateUnu.writeUTF(" Scriem acest text in fisier.");
            fisierDateUnu.writeInt(-1);
            fisierDateUnu.close();
            fisierUnu.close();
            System.out.println("Urmeaza citirea din fisier");
            // urmeaza citirea acestui fisier si afisarea datelor
            FileInputStream fisierDoi = new FileInputStream(
                "fisierUnu.out");
            DataInputStream fisierDateDoi = new DataInputStream(
                fisierDoi);
            byte b = fisierDateDoi.readByte();
            int i = fisierDateDoi.readInt();
            double d = fisierDateDoi.readDouble();
            String sir1 = fisierDateDoi.readUTF();
            String sir2 = fisierDateDoi.readUTF();
            System.out.println("b = " + b);
            System.out.println("i = " + i);
            i = fisierDateDoi.readInt();
            System.out.println("i = " + i);
            System.out.println("d = " + d);
            System.out.println("sir1 = " + sir1);
            System.out.println("sir2 = " + sir2);
            fisierDateDoi.close();
            fisierDoi.close();
        }
    }
}
```

```
catch (IOException e) {
    System.out.println("Eroare scriere fisier " + e);
}
}
```

Programul va afișa la execuție:

```
Urmeaza citirea din fisier
b = 127
i = 123456
i = -1
d = 12.34
sir1 = 123?
sir2 = Scriem acest text in fisier.
```

Pachetul java.io oferă și alte clase care se referă la fluxuri de nivel înalt. Constructorii fluxurilor de intrare de nivel înalt necesită ca argument un flux de nivel imediat mai jos. Acesta va fi sursa datelor citite de noul obiect. Similar, constructorii fluxurilor de ieșire de nivel înalt necesită un argument flux de nivel imediat mai jos. Noul obiect va scrie date către acest flux. Alte fluxuri de nivel înalt sunt:

- BufferedInputStream și BufferedOutputStream: aceste clase au buffere interne astfel încât oțeții pot fi citiți sau scriși în blocuri mari;
- PrintStream: această clasă este utilă la scrierea de text sau date primitive. Acestea sunt convertite la reprezentări caracter. Obiectele System.out și System.err sunt instanțe ale acestei clase;
- PushbackInputStream: această clasă permite ca ultimul octet citit să fie pus înapoi în flux ca și cum nu s-ar fi citit.

Exemplul 5.3.6. Puteam implementa un flux de intrare care citește dintr-un flux de intrare de tip buffer, care la rândul lui citește dintr-un flux fișier de intrare, astfel:

```
FileInputStream fis = new FileInputStream("fisier.in");
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

5.3.3.3. Fluxuri pentru citire și fluxuri pentru scriere

Fluxurile pentru citire (eng. *readers*) și cele pentru scriere (eng. *writers*) sunt ca și fluxurile de intrare și ieșire: cele de nivel jos comunică cu dispozitivele de intrare/ieșire, cele de nivel înalt comunică cu cele de nivel jos. Diferența dintre acestea este că fluxurile de citire și de scriere utilizează numai caractere Unicode.

Clasa FileReader este un exemplu de cititor de nivel jos, având cei mai utilizați constructori:

- FileReader(String cale);
- FileReader(File fisier);

Scriitorul asociat acestei clase este `FileWriter`, având constructorii:

- `FileWriter(String cale);`
- `FileWriter(File fisier);`

Alte clase de cititori și scriitori de nivel jos sunt:

- `CharArrayReader` și `CharArrayWriter`: citesc și scriu tablouri de caractere;
- `PipedReader` și `PipedWriter`: furnizează un mecanism pentru comunicarea între procese;
- `StringReader` și `StringWriter`: citesc și scriu `String`-uri.

Toți cititorii extind superclasa abstractă `Reader`. Această clasă oferă cele trei metode `read()` pentru citirea unui caracter, tablou de caractere sau a unui subșir de tablou de caractere. Atenție! Unitatea de informație vehiculată este acum `char` (reprezentat pe doi octeți) și nu `byte` (cum se întâmpla la fluxurile de până acum):

- `int read()` throws `IOException`;
Returnează următorul caracter din fișier (memorat în primii doi octeți nesemnificativi ai unei variabile de tip `int`) sau `-1`, dacă pointerul este la sfârșitul fișierului.
- `int read(char destinație[])` throws `IOException`;
Încearcă să citească suficiente caractere pentru a umple tabloul `destinație[]`. Returnează numărul de caractere citite din fișier sau `-1`, dacă pointerul este la sfârșitul fișierului.
- `abstract int read(char destinație[], int deplasament, int lungime)` throws `IOException`;
Încearcă să citească lungime caractere în tabloul `destinație[]`, începând de la deplasament. Returnează numărul de caractere citite din fișier sau `-1`, dacă pointerul este la sfârșitul fișierului.

Toți scriitorii extind superclasa abstractă `Writer`. Această clasă furnizează metode puțin diferite de cele trei bine cunoscute metode `write()`:

- `void write(int ch)` throws `IOException`;
Scrie caracterul `ch` care apare în cei mai semnificativi doi octeți ai lui `ch`.
- `void write(String str)` throws `IOException`;
Scrie `String`-ul `str`.
- `void write(char c[])` throws `IOException`;
Scrie caracterele din șirul `c[]`.
- `void write(String str, int deplasament, int lungime)` throws `IOException`;
Scrie lungime caractere din `String`-ul `str`, începând de la poziția deplasament.
- `void write(char destinație[], int deplasament, int lungime)` throws `IOException`;
Scrie lungime caractere din tabloul `destinație[]`, începând de la poziția deplasament.

Cititorii și scriitorii de nivel înalt se moștenesc din superclasele `Reader` sau `Writer`, deci au metodele listate mai sus. Ca și fluxurile de nivel înalt, când se construiește un cititor sau scriitor de nivel înalt, trebuie să trimitem un obiect din nivelul imediat următor. Clasele de nivel înalt sunt:

- `BufferedReader` și `BufferedWriter`: aceste clase au *buffere* interne, astfel încât datele pot fi citite sau scrise în blocuri mari. Sunt similare cu cele de la fluxuri;
- `InputStreamReader` și `OutputStreamReader`: aceste clase convertesc fluxuri de octeți și caractere Unicode. Implicit, clasele presupun că fluxurile folosesc codificarea implicită a caracterelor platformei;
- `LineNumberReader`: această clasă consideră intrarea ca fiind o secvență de linii de text. Apelul metodei `readLine()` returnează următoarea linie;
- `PrintWriter`: similară cu `PrintStream`, dar scrie caractere în loc de octeți;
- `PushbackReader`: similară cu `PushbackInputStream`, dar citește caractere în loc de octeți.

Exemplul 5.3.7. Următorul program pune în evidență utilizarea claselor `FileReader` și `LineNumberReader` pentru afișarea unui fișier text, în care fiecare linie este precedată de numărul ei.

```
import java.io.*;

public class LiniiNumerotate {
    public static void main(String args[]) {
        try {
            FileReader fr = new FileReader("LiniiNumerotate.java");
            LineNumberReader lnr = new LineNumberReader(fr);
            String s;
            while ((s = lnr.readLine()) != null) {
                System.out.println(lnr.getLineNumber() + ": " + s);
            }
            lnr.close();
            fr.close();
        } catch (IOException e) {
            System.out.println(
                "A aparut o exceptie de intrare/iesire");
        }
    }
}
```

După cum se observă, execuția acestui program va afișa codul sursă al programului de mai sus, punând la începutul fiecărei linii numărul ei, urmat de caracterul două puncte și spațiu.

Continuăm cu prezentarea în detaliu a două clase frecvent utilizate.

5.3.3.4. Clasele `PrintWriter` și `BufferedReader`

Începând cu JDK 1.1, sunt disponibile clasele `PrintWriter` și `BufferedReader`. Clasa `PrintWriter` are patru constructori:

- `public PrintWriter(Writer iesire);`
Creează un nou obiect `PrintWriter`, fără trimiterea automată a conținutului unei linii. Parametrul `iesire` este un flux orientat caracter.
- `public PrintWriter(Writer iesire, boolean autoFlush);`
Creează un nou obiect `PrintWriter`, cu trimiterea automată a conținutului unei linii (în cazul metodelor `println()`) dacă variabila `autoFlush` are valoarea `true`. Parametrul `iesire` este un flux orientat caracter.
- `public PrintWriter(OutputStream iesire);`
Creează un nou obiect `PrintWriter`, fără trimiterea automată a conținutului unei linii. Parametrul `iesire` este un flux de ieșire. Constructorul este util deoarece creează un obiect intermediar `OutputStreamWriter`, care va converti caracterele în octeți folosind codificarea implicită a caracterelor.
- `public PrintWriter(OutputStream iesire, boolean autoFlush);`
Creează un nou obiect `PrintWriter`, cu trimiterea automată a conținutului unei linii (în cazul metodelor `println()`). Parametrul `iesire` este un flux de ieșire. Constructorul este util deoarece creează un obiect intermediar `OutputStreamWriter`, care va converti caracterele în octeți folosind codificarea implicită a caracterelor.

Clasa `BufferedReader` are doi constructori:

- `public BufferedReader(Reader intrare, int marime);`
Creează un flux de intrare caracter asociat cititorului `intrare` care utilizează un buffer de dimensiune egală cu `marime`. Dacă valoarea lui `marime` este 0, atunci se aruncă o excepție `IllegalArgumentException`.
- `public BufferedReader(Reader intrare);`
Creează un flux de intrare caracter asociat cititorului `intrare` care utilizează un buffer de dimensiune implicită.

Pentru citirea și scrierea de linii de text, se pot utiliza metodele:

- `readLine()` din clasa `BufferedReader`. Aceasta citește o linie de text într-un șir (dacă dorim împărțirea acestui șir, folosim clasa `StringTokenizer`);
- `print()` și `println()` din clasa `PrintWriter`. Aceste metode scriu un șir într-un fișier (`println()` adaugă la sfârșitul fiecărei date scrise și caracterul `'\n'`).

Exemplul 5.3.8. Vom prezenta un program Java de sine stătător care ne permite să scriem caractere într-o arie text și apoi să salvăm textul într-un fișier. Vom declara un astfel de fișier ca atribut privat:

```
private PrintWriter fisierIesire;
```

Apoi creăm o instanță a clasei `PrintWriter` astfel:

```
fisierIntrare = new PrintWriter(new FileWriter("fisier.txt"), true);
```

Dacă fișierul `fisier.txt` nu există, atunci va fi creat. Dacă există, atunci va fi suprascris. S-a apelat constructorul clasei `FileWriter`, care permite specificarea numelui fișierului ca un șir de caractere. Acesta este transmis ca prim argument constructorului clasei `PrintWriter`. Cel de-al doilea parametru este o valoare booleană. Dacă este `true`, atunci buffer-ul intern va fi automat trimis (eng. *flushed*) în fișier la fiecare apel al metodei `print()`. În caz contrar, trebuie apelată explicit metoda `flush()`. În plus, dacă o parte ulterioară a execuției programului se blochează, atunci conținutul din buffer este pierdut. Pentru a scrie în fișier folosim metoda `print()`:

```
fisierIesire.print(intrareArieText.getText());
```

În final, închidem fișierul astfel:

```
fisierIesire.close();
```

Crearea unui fișier poate produce o excepție, deci aceasta trebuie verificată, fiind obligatorie „înconjurarea” codului într-un bloc `try-catch` pentru detectarea unei eventuale excepții de tip `IOException`. O verificare și mai riguroasă a posibilelor erori este cazul în care metoda `print()` detectează o eroare în timp ce scrie într-un fișier:

```
fisierIesire.print(intrareArieText.getText());
if (fisierIesire.checkError())
    System.err.println("Eroare in timpul scrierii in fisier");
```

Codul complet al aplicației mai sus discutate este următorul:

```
import java.io.*;
// importarea pachetelor pentru lucru cu obiecte grafice
import java.awt.*;
import java.awt.event.*;
```

```
public class Scieri extends Frame implements
    ActionListener, WindowListener {
    private TextArea intrareArieText;
    private Button butonSalvare;
    private PrintWriter fisierIesire;

    public static void main(String[] args) {
        Scieri f = new Scieri();
        f.setSize(200, 200);
        f.interfataGrafica();
        f.setVisible(true);
    }
}
```

```
// crearea elementelor grafice si
// adaugarea acestora la fereastra aplicatiei
public void interfataGrafica() {
    butonSalvare = new Button("save");
    add("North", butonSalvare);
    butonSalvare.addActionListener(this);
    intrareArieText = new TextArea(10, 50);
    add("Center", intrareArieText);
    addWindowListener(this); // pentru inchiderea ferestrei
}

// aceasta metoda este apelata cand s-a apasat
// butonul 'save'
public void actionPerformed(ActionEvent event) {
    if (event.getSource() == butonSalvare) {
        try {
            // salvarea textului in fisier
            fisierIesire = new PrintWriter(new FileWriter(
                "fisier.txt"), true);
            fisierIesire.println(intrareArieText.getText());
            fisierIesire.close();
        }
        catch (IOException e) {
            System.err.println("Eroare de fisier: " +
                e.toString());
            System.exit(1);
        }
    }
}

// aceasta metoda este apelata automat cand se apasa
// butonul pentru inchiderea ferestrei aplicatiei
public void windowClosing(WindowEvent event) {
    System.exit(0);
}

// acestea sunt metode din interfata WindowListener
// iar apelarea lor nu are nici un efect
public void windowIconified(WindowEvent event) {};
public void windowDeiconified(WindowEvent event) {};
public void windowOpened(WindowEvent event) {};
public void windowClosed(WindowEvent event) {};
public void windowActivated(WindowEvent event) {};
public void windowDeactivated(WindowEvent event) {};
}
```

Execuția programului Java de mai sus va implica apariția cadrului următor. După completarea ariei text, se poate apăsa butonul "save" și are loc salvarea sa în fișierul "fisier.txt".



Exemplul 5.3.9. Vom prezenta un program Java de sine stătător care ne permite să citim un fișier pentru a fi încărcat într-o arie text. La început, declarăm o instanță:

```
private BufferedReader fisierIntrare;

După care creăm o instanță:

fisierIntrare = new BuffererReader(new FileReader(umeFisier));

Apoi citim câte o linie din fișier, adăugând-o în aria text (utilizând metoda
append()). Apelăm metoda readLine() care întoarce null dacă s-a ajuns la sfârșitul
fișierului:

while ( (linie=fisierIntrare.readLine()) != null ) {
    arieTextIntrare.append(linie+"\n");
}

fisierIntrare.close();
```

Deci când utilizatorul apasă butonul load, atunci programul va executa următoarele acțiuni:

1. citește numele fișierului din câmpul text;
2. deschide fișierul cu acest nume;
3. citește linii din fișier și le adaugă la aria text, până la sfârșitul fișierului;
4. închide fișierul.

Codul complet al programului Java discutat este:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class Citiri extends Frame implements
```

```

ActionListener, WindowListener {
private TextArea arieTextIntrare;
private Button butonLoad;
private BufferedReader fisierIntrare;
private TextField numeCamp;

public static void main(String[] args) {
    Citiri f = new Citiri();
    f.setSize(300, 200);
    f.interfataGrafica();
    f.setVisible(true);
}

// crearea elementelor grafice si
// adaugarea acestora la fereastra aplicatie
public void interfataGrafica() {
    Panel top = new Panel();
    butonLoad = new Button("load");
    top.add(butonLoad);
    butonLoad.addActionListener(this);
    numeCamp = new TextField(20);
    top.add(numeCamp);
    numeCamp.addActionListener(this);
    add("North", top);
    arieTextIntrare = new TextArea("", 10, 50);
    add("Center", arieTextIntrare);
    addWindowListener(this); // pentru inchiderea ferestrei
}

// metoda se va apela la apasarea butonului 'load'
public void actionPerformed(ActionEvent event) {
    if (event.getSource() == butonLoad) {
        String numeFisier;
        numeFisier = numeCamp.getText();
        try {
            fisierIntrare = new BufferedReader(new FileReader
                (numeFisier));
            arieTextIntrare.setText(""); // stergem continutul
            //ariei text
            String line;
            // citim cate o linie din fisier si o adaugam la aria text
            while ((line = fisierIntrare.readLine()) != null) {
                arieTextIntrare.append(line + "\n");
            }
            fisierIntrare.close();
        }
    }
}

```

```

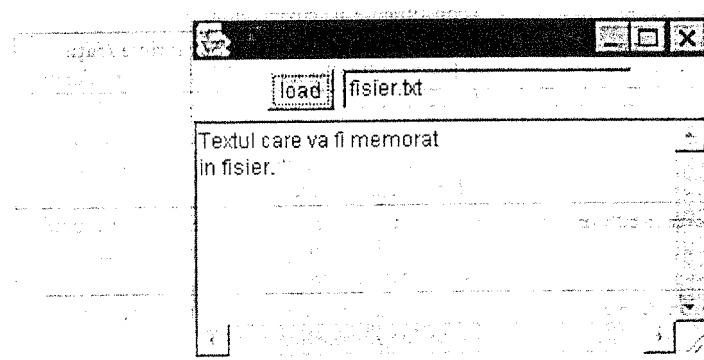
    }
    catch (IOException e) {
        System.err.println("Eroare in fisierul: " + numeFisier +
            ": " + e.toString());
    }
}

// aceasta metoda se va apela la apasarea butonului 'Close'
// pentru inchiderea ferestrei aplicatiei
public void windowClosing(WindowEvent event) {
    System.exit(0);
}

// aceste metode sunt definite in interfata WindowListener
// iar apelul acestora nu are nici un efect
public void windowIconified(WindowEvent event) {};
public void windowDeiconified(WindowEvent event) {};
public void windowOpened(WindowEvent event) {};
public void windowClosed(WindowEvent event) {};
public void windowActivated(WindowEvent event) {};
public void windowDeactivated(WindowEvent event) {};
}

```

Iată o posibilă execuție a acestui program Java:



5.3.3.5. Clasa StreamTokenizer

Clasa `java.io.StreamTokenizer` oferă suport pentru parsarea datelor dintr-un flux de intrare. Aceasta extrage câte o entitate (eng. *token*) și poate recunoaște valorile numerice, șiruri de caractere date între simbolurile ghilimele, identificatori și comentarii.

Clasa amintită posedă un singur constructor:

- `public StreamTokenizer(Reader r)`

unde `r` este fluxul de intrare de tip `Reader` care va fi parsat.

Pentru ușurința indentificării elementelor și lucrul cu acestea, clasa `StreamTokenizer` pune la dispoziție următoarele date membre:

Denumirea atributului	Descrierea atributului
<code>public double nval</code>	Dacă elementul curent este un număr, atunci valoarea acestuia se va regăsi în acest atribut.
<code>public String sval</code>	Dacă elementul curent este de tip șir de caractere, atunci șirul corespunzător va fi stocat în acestă dată membră.
<code>public static final int TT_EOF</code>	Constantă care indică faptul că fluxul de intrare a ajuns la sfârșit.
<code>public static final int TT_EOL</code>	Constanta arată că s-a citit sfârșitul unei linii.
<code>public static final int TT_NUMBER</code>	Constanta indică întâlnirea unui număr.
<code>public static final int TT_WORD</code>	Constanta semnalează citirea unui cuvânt.
<code>public int ttype</code>	Indică tipul de element depistat și poate lua ca valoare una din constantele prezentate mai sus.

Pentru a vedea utilitatea clasei `StreamTokenizer`, vom urmări facilitățile date de metodele acesteia:

Prototipul metodei	Descrierea metodei
<code>public void wordChars(int min, int max)</code>	Stabilește intervalul în care se găsesc caracterele constituente ale cuvinetelor...
<code>public void whitespaceChars(int min, int max)</code>	Indică intervalul în care sunt cuprinse spațiile albe. Spațiile albe nu vor fi luate în considerație.
<code>public void ordinaryChars(int min, int max)</code>	Stabilește interval pentru caracterele ordinare (cele care sunt considerate <i>token-uri</i>). Când se întâlnește un astfel de caracter, va fi considerat de tip șir de caractere.
<code>public void commentChar(int c)</code>	Stabilește caracterul care indică începutul unui comentariu ce ține până la sfârșitul liniei. Comentariile sunt ignorate.
<code>public void quoteChar(int c)</code>	Fixează caracterul utilizat pentru încadrarea șirurilor de caractere.
<code>public void parseNumbers()</code>	Indică faptul că numerele vor fi identificate, iar valoarea lor se va regăsi în atributul <code>nval</code> .
<code>public void eolIsSignificant(boolean flag)</code>	Stabilește dacă sfârșitul de linie prezintă sau nu importanță. În caz afirmativ, la întâlnirea acestuia atributul <code>ttype</code> va lua valoarea constantei <code>TT_EOL</code> .

Prototipul metodei	Descrierea metodei
<code>public void slashStarComments(boolean flag)</code>	Specifică dacă textul cuprins între <code>/*</code> și <code>*/</code> este considerat comentariu.
<code>public void slashSlashComments(boolean flag)</code>	Specifică dacă textul cuprins între <code>//</code> și sfârșitul de linie este considerat comentariu.
<code>public void lowerCaseMode(boolean fl)</code>	Stabilește dacă textul va fi convertit automat la minuscule.
<code>public int nextToken() throws IOException</code>	Citește următorul element (<i>token</i>) și îi identifică tipul. Returnează valoarea câmpului <code>ttype</code> , iar în cazul în care apar probleme la citirea din flux, va fi aruncată o excepție de tip <code>IOException</code> .
<code>public void pushBack()</code>	Pune înapoi elementul citit în flux (la început) fără a modifica valorile atributelor.
<code>public int lineno()</code>	Returnează numărul liniei curente.
<code>public String toString()</code>	Returnează șirul de caractere corespunzător elementului curent.

Exemplul 5.3.10. Programul Java va citi datele din fișierul `fișier.txt`. Acesta are următorul conținut:

```
# comentariu
Ioana are 125.75 EURO.
/* comentariu
pe mai multe randuri */
```

Se vor parsea datele și vor fi identificate numerele și cuvintele. Codul programului este următorul:

```
import java.io.*;

public class Identificare {
    static public void main(String arg[]) {
        StreamTokenizer parser;
        try {
            /** crearea unui flux de intrare dintr-un fisier
             care va final parsat de un StreamTokenizer */
            parser = new StreamTokenizer(new FileReader(
                "fisier.txt"));
            /** Caracterul '#' va stabili un comentariu
             pana la sfarsitul liniei */
            parser.commentChar('#');
            /* Vor fi permise comentariile asemeni acestuia */
            parser.slashStarComments(true);
            /** Sfarsitul de linie va final ignorat */
            parser.eolIsSignificant(false);
```



```

/** citirea din flux si parsarea acestuia */
while(parser.nextToken()!=parser.TT_EOF)
    /** daca am intalnit un numar */
    if (parser.ttype == parser.TT_NUMBER)
        System.out.println("Numar: " + parser.nval);
    /** daca s-a identificat un sir de caractere */
    else if (parser.ttype == parser.TT_WORD)
        System.out.println("Cuvant: " + parser.sval);
    /** daca am intalnit sfarsitul unei linii */
    else if (parser.ttype == parser.TT_EOL)
        /** mai sus am stabilit ignorarea sfarsiturilor de
        linie */
        System.out.println("Sfarsit de linie");
    /** tratarea exceptiei */
    catch(IOException e) {
        System.err.println("Eroare de intrare/iesire: " + e);
    }
}

```

Execuția programului anterior va afișa la consolă:

```

Cuvant: Ioana
Cuvant: are
Numar: 125.75
Cuvant: EURO.

```

Observăm că valoarea numerică a fost identificată, iar comentariile și sfârșiturile de linie au fost ignorate.

Exemplul 5.3.11. Vom prezenta un program Java de sine stătător care ne permite să căutăm anumite informații în fișier. Presupunem că avem un fișier cu linii de forma:

```

nume, nota1, nota2 <SfarsitLinie>

```

Din punct de vedere al interfeței grafice, avem cinci câmpuri text, unul de afișare a mesajului de citire a numelui fișierului, două câmpuri editabile pentru citirea numelui fișierului și celălalt pentru citirea numelui persoanei ale cărei note trebuie căutate în fișierul deschis pentru citire. Mai există două câmpuri text pentru afișarea celor două note găsite. Începerea căutării se semnalează ca eveniment când se apelează butonul Cautare. Se presupune că numele persoanei este unic în baza de date. Spre deosebire de programul precedent, trebuie să procesăm linia curentă folosind un obiect de tip `java.util.StringTokenizer`. La parcurgerea fișierului se procedează aproximativ astfel (pseudocod):

```

boolean gasit = false;
while ( (exista linii de citit) && (not gasit)) {

```

```

citeste prima inregistrare;
if (primul camp coincide cu numele) {
    gasit = true;
    afiseaza notele in ariile text
}
}

```

Codul complet al programului Java asociat este:

```

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*; // pachet necesar pentru clasa
// StringTokenizer

```

```

public class CautareFisier extends Frame implements
    ActionListener, WindowListener {
    // declararea datelor membre
    private BufferedReader fisierIntrare;
    private Button butonCautare;
    private TextField rezultat1;
    private TextField rezultat2;
    private TextField campPersoana;
    private TextField campNumeFisier;
    private TextField campEroare;
    private String numeFisier;

```

```

public static void main(String[] args) {
    CautareFisier f = new CautareFisier();
    f.setSize(300, 150);
    f.interfataGrafica();
    f.setVisible(true);
}

```

```

public void interfataGrafica() {
    setLayout(new FlowLayout());
    campEroare = new TextField("Dati numele fisierului: ");
    campEroare.setEditable(false);
    add(campEroare);
    campNumeFisier = new TextField(20);
    campNumeFisier.setText("");
    add(campNumeFisier);
    butonCautare = new Button(" Cautare ");
    add(butonCautare);
    butonCautare.addActionListener(this);
    add(new Label("Numele: "));
}

```

```

campPersoana = new TextField(20);
campPersoana.setText("");
add(campPersoana);
add(new Label("Nota 1:"));
rezultat1 = new TextField(5);
rezultat1.setEditable(false);
add(rezultat1);
add(new Label("Nota 2:"));
rezultat2 = new TextField(5);
rezultat2.setEditable(false);
add(rezultat2);
this.addWindowListener(this); // pentru inchiderea ferestrei
}

public void actionPerformed(ActionEvent ev) {
    if (ev.getSource() == butonCautare) {
        numeFisier = campNumeFisier.getText();
        try {
            fisierIntrare = new BufferedReader(new FileReader(
                numeFisier));
        }
        catch (IOException e) {
            System.err.println("Nu gasim fisierul: " + numeFisier +
                ": " + e.toString());
            return;
        }
        try {
            String linie;
            boolean gasit = false;
            while (((linie = fisierIntrare.readLine()) != null) &&
                (! gasit)) {
                StringTokenizer tokens = new StringTokenizer(
                    linie, " ,");
                String numeFisierIntrare = tokens.nextToken();
                if (campPersoana.getText().equals(
                    numeFisierIntrare)) {
                    gasit = true;
                    rezultat1.setText(tokens.nextToken());
                    rezultat2.setText(tokens.nextToken());
                }
            }
            fisierIntrare.close();
        }
    }
}

```

```

catch (IOException e) {
    System.err.println("Eroare in fisierul: " + numeFisier +
        ": " + e.toString());
    System.exit(1);
}
}

public void windowClosing(WindowEvent ev) {
    System.exit(0);
}

public void windowIconified(WindowEvent ev) {};
public void windowDeiconified(WindowEvent ev) {};
public void windowOpened(WindowEvent ev) {};
public void windowClosed(WindowEvent ev) {};
public void windowActivated(WindowEvent ev) {};
public void windowDeactivated(WindowEvent ev) {};
}

```

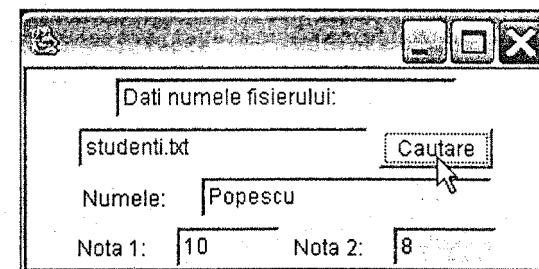
Presupunem că avem fișierul `studenti.txt`, astfel:

```

Popescu, 10, 8
Ionescu, 6, 7
Pop, 9, 7

```

Atunci o execuție a programului de mai sus poate fi:



Atenție! A nu se face confuzie între clasele `java.io.StreamTokenizer`, care parsează un flux de intrare, și `java.util.StringTokenizer`, care parsează un șir de caractere. În plus, clasa `StreamTokenizer` oferă suport pentru identificarea valorilor numerice, ignorarea comentariilor etc.

5.3.3.6. Clasa System

Această clasă pune la dispoziție trei fluxuri predefinite: `System.in`, `System.out` și `System.err`. Aceste fluxuri sunt cunoscute sub denumirea de ecran *terminal* sau

consola de intrare/ieșire. Fluxul `System.in` poate fi folosit direct pentru citirile de la tastatură și pentru citirile redirectate din comenzi ale sistemului de operare. Totuși, metodele puse la dispoziție sunt de nivel jos, spre deosebire de citirea clasică dintr-un șir de caractere. Recomandăm utilizarea constructorului `BufferedReader` și metoda `readLine()`, astfel:

```
private BufferedReader tastatura;
tastatura = new BufferedReader(new InputStreamReader(System.in), 1);
```

Spre deosebire de precedentele apeluri ale constructorului `BufferedReader`, acum avem doi parametri, al doilea fixând lungimea buffer-ului la 1, eliminând astfel „efectul de buffer”. Constructorul `BufferedReader` (fără precizarea lungimii buffer-ului) așteaptă umplerea buffer-ului său (cu mai multe linii de text). Doar când buffer-ul este plin, atunci va transmite prima linie către `readLine()`. Efectul este că utilizatorul trebuie să introducă mai multe linii de text pentru ca prima linie să fie accesată de către program. De aceea am setat lungimea buffer-ului la 1. Astfel `readLine()` va citi textul introdus imediat după tastarea lui <ENTER>. O dată creat un flux, citirea din el se realizează astfel:

```
String linie = tastatura.readLine();
```

Exemplul 5.3.12. Vom prezenta un program Java de sine stătător care ne permite să citim un `String` de la tastatură.

```
import java.io.*;

public class CitiriTastatura {
    private BufferedReader tastatura;

    public static void main(String[] args) {
        CitiriTastatura obiect = new CitiriTastatura ();
        obiect.executa();
    }

    private void executa() {
        tastatura = new BufferedReader(new InputStreamReader(
            System.in), 1);
        System.out.print("Dati un sir:");
        try {
            System.out.flush(); // golește bufferul de iesire la consola
            String linie = tastatura.readLine();
            System.out.println("Ați tastat " + linie);
        }
        catch (IOException e) {
            System.out.println("Intrare de la tastatura " +
                e.toString());
        }
    }
}
```

```
System.exit(2);
```

Fluxul `System.out` are la dispoziție metodele `print()`, `println()` pentru afișarea unui `String`. Comanda `System.out.flush()` asigură că textul care a fost argument pentru `print()` va fi imediat afișat (nu este și cazul lui `println()`, unde nu este nevoie de `flush()`). Fluxul `System.err` poate fi utilizat similar cu `print()` și `println()`. Mesaje de eroare apar pe ecranul consolei. De exemplu, putem avea un apel de genul:

```
System.err.println("Eroare !!! Fisierul nu a fost găsit");
```

Clasa `System` pune la dispoziție o metodă pentru terminarea imediată a aplicațiilor. Este vorba despre `System.exit()`. Această funcție are un argument întreg. Convenția de ieșire normală este 0, în timp ce numere diferite de 0 înseamnă că a avut loc o eroare și programul nu s-a terminat normal. De exemplu, putem avea mesaje:

```
System.exit(0); //iesire cu succes
```

```
System.exit(3); //iesire cu cod eroare 3
```

Codul întors la terminarea execuției unui program poate fi utilizat pentru a determina dacă acesta s-a executat cu succes sau pentru a stabili natura erorii apărute.

Exemplul 5.3.13. Vom prezenta un program Java de sine stătător care permite căutarea unui `String` într-un fișier. Numele fișierului din care se face citirea va fi furnizat ca argument în linia de comandă și memorat în `args[0]` din metoda `main(String args[])`. Considerăm că fișierul sursă al aplicației care realizează căutarea se numește `Gaseste.java`. Atunci comanda de execuție a aplicației poate fi:

```
java Gaseste fis.in
```

unde `fis.in` este numele fișierului de intrare. Apoi citim de la tastatură cuvântul pe care dorim să-l căutăm în respectivul fișier. Vom obține liniile care conțin acel cuvânt, plus o linie precedentă, cât și una ulterioară acelei linii în caz afirmativ. Programul afișează toate aparițiile cuvântului. În cazul în care cuvântul nu este găsit, nu se va afișa nimic. Codul complet al programului Java care implementează cele discutate mai sus este:

```
import java.io.*;
```

```
class Gaseste {
    private String linia1, linia2, linia3;
    private BufferedReader tastatura, fluxIntrare;
```

```
public static void main(String[] args) {
    Gaseste gaseste = new Gaseste();
```

```

    gaseste.cauta(args[0]);
}

// metoda care realizeaza cautarea efectiva
private void cauta(String numeFisier) {
    tastatura = new BufferedReader(new InputStreamReader(
        System.in), 1);
    /* se apeleaza metoda prompt() care intoarce sirul citit
    de la tastatura */
    String sir = prompt("Dati sirul ce trebuie cautat:");
    linia1 = "";
    linia2 = "";
    try {
        fluxIntrare = new BufferedReader(new FileReader(
            numeFisier));
        /* in linia3 se va citi cate o linie din fisierul de
        intrare */
        while ((linia3 = fluxIntrare.readLine()) != null) {
            /* daca linia2 contine sirul citit de la tastatura
            se va afisa linia in care apare (linia2) si
            contextul (o linie inainte si una dupa) */
            if (linia2.indexOf(sir) >= 0)
                afiseazaLinia();
            // se pastreaza ultimele doua linii citite
            linia1 = linia2;
            linia2 = linia3;
        }
        // am ajuns la sfarsitul fisierului
        linia3 = "";
        // testam daca sirul cautat apare pe ultima linie
        if (linia2.indexOf(sir) >= 0)
            afiseazaLinia();
        // inchidem fisierul de intrare
        fluxIntrare.close();
    } catch (IOException e) {
        System.err.println("Eroare la cautarea sirului: " +
            e.toString());
        System.exit(1);
    }
}

// aceasta metoda afiseaza linia gasita si contextul in care apare
private void afiseazaLinia() {

```

```

    System.out.println("<<— context:");
    System.out.println(linia1);
    System.out.println(linia2);
    System.out.println(linia3);
    System.out.println("
    —>>");
    System.out.println("");
}

/* aceasta metoda afiseaza sirul de caractere primit (mesaj)
si citeste un sir de caractere de la tastatura */
private String prompt(String mesaj) {
    String raspuns = "";
    try {
        System.out.print(mesaj);
        System.out.flush();
        raspuns = tastatura.readLine();
    } catch (IOException e) {
        System.err.println("Eroare la citirea de la tastatura: "
            + e.toString());
        System.exit(2);
    }
    return raspuns;
}
}

```

5.4. Serializarea obiectelor

O caracteristică importantă a aplicațiilor în rețele este necesitatea de a trimite și primi date între două aplicații, posibil între mașini aflate la distanță sau de a salva starea curentă a obiectelor și restaurarea acestora la o execuție ulterioară. Acest lucru se realizează în trei faze:

1. împachetarea datelor (*serializare*);
2. trimiterea/salvarea datelor;
3. despachetarea datelor (*deserializare*).

1. *Serializarea* este procesul de împărțire a datelor în octeți astfel încât să poată fi trimiși printr-un flux de ieșire. Utilizând fluxuri normale, un întreg se transmite împărțindu-l în patru octeți și scriindu-l într-un obiect `OutputStream`. Serializarea datelor de tipuri primitive se realizează cu ajutorul clasei `DataOutputStream`. Dacă dorim să trimitem date complexe reprezentând un obiect, trebuie să împărțim obiectul într-o formă astfel încât să poată fi reconstruit ulterior. Dacă obiectul referă alte obiecte, atunci procesul poate fi complex.

Se restaurează o copie a obiectului original și o listă de obiecte care-l referă, chiar dacă mașina virtuală este aceeași. Cu toate acestea, dacă un obiect este transmis de două ori în același flux, atunci va reface două referințe la aceeași copie. Obiectul serializabil implementează o interfață pentru a rezolva aceasta la un nivel înalt, preluând din sarcina programatorului serializarea obiectelor (permițând totuși flexibilitatea trimiterii obiectelor).

2. Trimiterea/salvarea datelor se referă la procesul ajungerii datelor de la sursă la destinație, respectiv din program într-un fișier extern. Fluxurile obiect Java furnizează facilități de serializare a obiectelor de-a lungul fluxurilor orientate octet, deci se pot accesa toate mecanismele de transport existente.
3. *Deserializarea* se referă la despachetarea datelor. Procesul de serializare reduce datele într-un șir de octeți, dar păstrează suficiente informații pentru reconstrucția ulterioară a datelor.

Serializarea și deserializarea se realizează în Java cu ajutorul claselor `ObjectOutputStream` și `ObjectInputStream`.

5.4.1. Clasa `ObjectOutputStream`

Clasa `ObjectOutputStream` implementează interfața `ObjectOutput` (derivată din interfața `DataOutput`) și are toate metodele definite (nici una nu este abstractă). Ca un `FilterOutputStream`, acesta se atașează unui `OutputStream` existent în maniera cunoscută.

Clasa `ObjectOutputStream` are doar un constructor:

- `ObjectOutputStream(OutputStream ieseire) throws IOException;`
Acesta creează un obiect `ObjectOutputStream` atașat argumentului `ieseire`. Constructorul poate arunca o excepție `IOException`, deoarece în fluxul atașat se scrie imediat un antet (eng. *header*).

Exemplul 5.4.1. După cele discutate mai sus, putem avea mai multe forme ca argument pentru constructor:

```
FileOutputStream unFlux = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(unFlux);

sau

// out este un camp din clasa FilterOutputStream declarat astfel
// protected OutputStream out;
ObjectOutputStream oos = new ObjectOutputStream(out);
```

Când sunt serializate tipuri primitive, atunci se pot utiliza metode ale clasei `DataOutput`. Dintre metodele clasei `ObjectOutputStream` enumerăm pe cele mai importante:

- `void writeObject(Object o) throws IOException;`
Este responsabilă de serializarea obiectelor și trimiterea lor către flux.

Exemplul 5.4.2. Continuând notațiile din exemplele de mai sus, avem:

```
int [] sir = new int[10];
oos.writeObject(sir);
Color culoare = new Color(240, 240, 240);
oos.writeObject(culoare);
```

Metoda `writeObject()` poate trimite toate câmpurile unui `Object`, inclusiv câmpurile private și `protected`. Doar câmpurile marcate `transient` sau `static` nu vor fi trimise. Din cauza acestor implicații de securitate, o clasă trebuie explicit să permită accesul să fie serializată prin implementarea interfeței `Serializable` sau `Externalizable`. Încercarea de a serializa un obiect care nu implementează una din aceste interfețe va implica o excepție `NotSerializableException`.

- `void reset() throws IOException;`
Resetează obiectul `ObjectOutputStream` către o stare inițială, ca atunci când a fost creat. Fiecare obiect `ObjectOutputStream` păstrează urma fiecărui obiect (din clasa `Object`) care a fost trimis. Astfel, dacă același obiect este scris către un `ObjectOutputStream` de două ori, atunci a doua oară se va scrie o referință înapoi, chiar dacă obiectul original a fost modificat. Acest lucru este important, deoarece chiar dacă au fost făcute referințe între obiecte de-a lungul apelurilor metodei `writeObject()`, se va putea reconstitui structura de referințe inițială. După un apel al lui `reset()`, dacă un obiect se va retrimite, atunci se va trimite o nouă copie.
- `void defaultWriteObject() throws IOException;`
Scrie obiectul apelant (obiectul care apelează aceasta metodă) către un `ObjectOutputStream`. Această metodă este utilizată doar în implementările obișnuite ale lui `writeObject()`.
- `protected Object replaceObject(Object obiect) throws IOException;`
Este utilizată de subclasele `ObjectOutputStream` pentru înlocuirea anumitor obiecte cu altele înaintea transmisiei. Acest lucru este folositor când se dorește serializarea unei clase declarate `final` care nu implementează interfața `Serializable`. În schimb, se poate înlocui obiectul care trebuie trimis cu un alt obiect care se poate transmite și care ulterior este convertit înapoi la original. Această reconversie este rezolvată de metoda `resolveObject()` a clasei `ObjectInputStream`. Fiecare obiect care este scris într-un `ObjectOutputStream` va fi trimis acestei metode. Pentru a-l lăsa neschimbat, se returnează obiect, altfel o valoare echivalentă serializabilă.
- `protected boolean enableReplaceObject(boolean permisiune) throws SecurityException;`
Deoarece metoda `replaceObject()` introduce anumite probleme, aceasta nu este inițial permisă. Pentru ca înlocuirea să fie permisă, este necesar întâi apelul metodei `enableReplaceObject()`; aceasta va permite (sau nu) înlocuirea obiectului conform cu parametrul `permisiune`.

- `protected void annotateClass(Class clasa) throws IOException;`
Este utilizată de subclasele `ObjectOutputStream` pentru a scrie informații suplimentare la transmiterea clasei `clasa`. Aceasta va apărea prima dată când un obiect din această clasă este transmis prin flux. Fluxurile de obiecte de obicei specifică numele clasei și informații despre versiune: o subclasă poate, de exemplu, să adauge locația clasei codului binar Java.
- `protected void annotateProxyClass(Class clasa) throws IOException;`
Începând cu JDK 1.3, subclasele pot implementa această metodă pentru memorarea datelor obișnuite în flux împreună cu descriptorii pentru clasele proxy dinamice. Metoda se apelează exact o dată pentru fiecare descriptor de clasă din flux. Metoda corespunzătoare din `ObjectInputStream` este `resolveProxyClass()`.

Anumite obiecte pot refuza să fie serializate din motive de securitate și vor arunca excepția `NotSerializableException` dacă se încearcă serializarea lor. În particular, când se serializează un obiect, orice câmp `private` este expus. Aceasta prezintă un risc de securitate, deoarece aceste câmpuri pot fi extrase din fluxul de transmisie a octeților. Din acest motiv, Java necesită ca toate clasele serializabile să declare explicit acest lucru.

Mecanismul de declarare a permisiunii de serializare este implementarea interfeței `Serializable` sau `Externalizable` și care să implementeze orice metodă cerută. Marea majoritate a nucleului claselor API suportă serializarea prin implementarea interfeței `Serializable` (cum ar fi: `String`, `Rectangle`, `Components` etc), dar nu și clase cum ar fi `InputStream` sau `Socket`.

Există multe alte excepții de tip `IOException` care pot fi aruncate de metoda `writeObject()`. Dacă apare o excepție în timpul unui apel al lui `writeObject()`, atunci vor fi serializate niște date speciale, astfel încât la restaurare se va primi o excepție `WriteAbortedException` când se va încerca citirea obiectului.

5.4.2. Clasa `ObjectInputStream`

Clasa `ObjectInputStream` implementează interfața `ObjectInput` (derivată din `DataInput`) și are toate metodele definite (în particular, `readObject()`). Ca un `FilterInputStream`, constructorul atașează unui obiect `InputStream` existent, de unde toate datele sunt citite. Tipurile primitive sunt citite într-un mod normal, dar există metode noi pentru lucrul cu tipuri referință.

Clasa `ObjectInputStream` furnizează un constructor public și unul `protected`:

- `ObjectInputStream(InputStream in) throws IOException;`
Creează un obiect `ObjectInputStream` atașat obiectului `in`. Acest constructor poate arunca o excepție `IOException`, deoarece este citit imediat un antet din fluxul atașat pentru a asigura compatibilitatea cu obiectul de tip `ObjectOutputStream` utilizat la scrierea datelor.

- `protected ObjectInputStream() throws IOException, SecurityException;`
Permite subclaselor înlocuirea protocolului de serializare implicit.

Când sunt citite tipuri primitive, pot fi utilizate metodele interfeței `DataInput`.

Exemplul 5.4.3. Iată un cod Java pentru citirea a patru octeți și reconstituirea unui întreg:

```
ObjectInputStream in = new ObjectInputStream(
    socket.getInputStream());
int valoare;
value = in.readInt();
```

În plus față de metodele clasei `DataInput`, clasa `ObjectInputStream` furnizează metode relative la procesul de deserializare. Acestea includ metode pentru rezolvarea fișierelor de clasă necunoscute. Metodele public furnizate sunt toate descrise de interfața `ObjectInput`, care extinde `DataInput` adăugând și alte metode relativ la obiecte ca și cele din clasa `InputStream`. Iată care sunt cele mai importante metode din această clasă:

- `Object readObject() throws IOException, ClassNotFoundException;`
Este responsabilă pentru deserializarea obiectelor dintr-un flux de comunicare.

Exemplul 5.4.4. Subprogramul Java de mai jos citește trei obiecte din clasele `Color`, `String` și `Date`. Metoda `readObject()` returnează un `Object` care se convertește la un tip convenabil. Transmiterea unui obiect `Class` în fluxul de obiecte nu înseamnă trimiterea codului binar Java al clasei respective. De fapt, se transmite doar numele clasei și semnatura (adică membrii acesteia și câți octeți ocupă fiecare).

```
Color culoare = (Color) in.readObject();
String sir = (String) in.readObject();
Date azi = (Date) in.readObject();
```

Clasa `ObjectInputStream` furnizează protecție împotriva încercărilor de citire greșite cum ar fi citirea de date binare dintr-un flux utilizând metode uzuale de tip `read()` când următorul articol este un `Object` sau citirea unui `Object` serializat când următorul articol este o dată binară. Încercarea de citire a unei date normale, când următorul articol în flux este un `Object`, va returna `EOF`. Încercarea de citire a unui `Object`, când următorul articol din flux este o dată normală, va implica o excepție `OptionalDataException`.

- `void defaultReadObject() throws IOException, ClassNotFoundException;`
Citește câmpurile obiectului apelat din obiectul `ObjectInputStream`. Acesta este folosit doar de implementarea obișnuită a lui `readObject()`. Apelul acestei metode într-un mod neadecvat conduce la o excepție `NotActiveException`.

- `protected Object resolveObject(Object ob) throws IOException;`
Este utilizată de subclasele lui `ObjectInputStream` pentru a înlocui anumite obiecte cu altele după recepție. Aceasta este folosită de obicei în asociere cu metoda `replaceObject()` a unui obiect de tip `ObjectOutputStream` pentru a permite transmiterea de obiecte care nu sunt de obicei serializabile. Aceasta metodă trebuie să returneze un obiect (`Object`) care este compatibil cu clasa originală (adică o instanță a clasei originale sau o subclasă).
De exemplu, un obiect `FileInputStream` poate fi înlocuit pentru transmisie cu un obiect `SerializedFileInputStream` care include numele fișierului asociat. După recepție, metoda `resolveObject()` poate înlocui obiectul cu un nou `FileInputStream`.
- `protected boolean enableResolveObject(boolean permisiune) throws SecurityException;`
Deoarece mecanismul de înlocuire introduce anumite probleme de securitate relativ la accesul la datele private, acesta nu este inițial permis. Pentru a permite rezolvarea obiectului de către un `ObjectInputStream`, este necesar să apelăm metoda `enableResolveObject()`; aceasta va permite (sau nu) înlocuirea obiectului, conform cu argumentul permisiune.
- `protected Class resolveClass(ObjectStreamClass osc) throws IOException, ClassNotFoundException;`
Este folosită de subclasele lui `ObjectInputStream` pentru a citi informații suplimentare scrise de metoda `annotateClass()` a obiectului de tip `ObjectOutputStream` și apoi să rezolve clasa respectivului obiect.
De exemplu, un obiect `ObjectOutputStream` poate transmite codul actual al unei clase (sau locul unde se găsește) și un `ObjectInputStream` poate atunci utiliza acestea pentru crearea clasei dacă aceasta nu este deja disponibilă local.

În ceea ce privește excepțiile fluxului de obiecte, acestea pornesc de la `IOExceptions` până la accesul fluxului suport sau `ObjectStreamException` (o subclasă a lui `IOException`) relativ la procesul de serializare. Excepțiile `ObjectStreamException` sunt:

- `InvalidClassException` – indică faptul că o clasă locală nu poate fi utilizată pentru deserializarea unui obiect dintr-un flux, chiar dacă obiectul serializat aparține unei clase care este prezentă pe calculatorul local;
- `NotSerializableException` – poate fi aruncată în timpul citirii sau scrierii și indică faptul că obiectul care s-a citit sau scris nu suportă serializarea;
- `StreamCorruptedException` – indică faptul că un flux nu conține date flux obiecte valide sau că informația din antetul fluxului este incorectă;
- `NotActiveException` – este aruncată, dacă una din metodele de serializare a fluxului de obiecte obișnuit este accesată de oriunde, numai că nu de o implementare de serializare obișnuită;
- `InvalidObjectException` – este aruncată, dacă un obiect eșuează după testul de validare de după scriere;

- `OptionalDataException` – este aruncată, dacă s-a făcut o încercare de citire a unui obiect dintr-un flux care conține date primitive;
- `WriteAbortedException` – este aruncată la încercarea de citire a unui obiect care a eșuat să fie scris, pentru că a apărut o excepție în timpul procesului de scriere.

5.4.3. Interfața `Serializable`

Interfața `Serializable` este o interfață de marcare utilizată de clase pentru a indica că acestea pot fi serializate prin mecanismul implicit. Această interfață nu are date sau metode! Este doar un marcator (eng. *flag*) care informează fluxurile de obiecte că instanțele serializate ale unei clase nu vor dezvălui informații sensibile sau deschise la problemele de securitate posibile.

Dacă o subclasă a unei clase neserializabile dorește să implementeze interfața `Serializable`, superclasa imediat următoare trebuie să furnizeze un constructor care nu are nici un argument. Este responsabilitatea subclasei să serializeze toate datele superclasei folosind următoarele metode obișnuite:

- `private void writeObject(ObjectOutputStream ieseire) throws IOException;`
Va fi apelată când o instanță a acestei clase este serializată către obiectul ieseire. Ar trebui apelată metoda `defaultWriteObject()` pentru ieseire înaintea scrierii datelor obișnuite proprii folosind metodele standard `ObjectOutputStream`, chiar dacă nu posedă date care vor fi serializate de mecanismul implicit. Aceasta va asigura compatibilitatea cu versiuni viitoare ale clasei care doresc să aibă avantajele opțiunii de serializare implicită;
- `private void readObject(ObjectInputStream intrare) throws IOException, ClassNotFoundException;`
Va fi apelată când o instanță a acestei clase este deserializată din obiectul intrare. Ar trebui apelată metoda `defaultReadObject()` pentru intrare înaintea citirii datelor obișnuite proprii și executării oricărei inițializări a câmpurilor *transient*.

Continuăm cu metode de rezoluție și înlocuire. Aceste metode pot fi furnizate de obiecte serializabile care doresc să conducă forma lor serializată către altă clasă. Aceasta este foarte utilă, deoarece separă detaliile de serializare de implementarea actuală a clasei. Acest lucru este important când detaliile de serializare au devenit implicate din probleme de versiuni. Aceste metode au apărut în versiunea JDK 1.2 și sunt:

- `private Object writeReplace() throws ObjectStreamException;`
Ar trebui să returneze o copie a acestui obiect pentru scopuri de serializare. Dacă o clasă care implementează această metodă este scrisă pentru un obiect `ObjectOutputStream`, atunci această metodă va fi apelată și va returna obiectul serializat (care trebuie să fie serializabil). De obicei, aceasta va implementa următoarea metodă `readResolve()` pentru ca obiectul să poată fi translatat înapoi către tipul original în vederea deserializării;

- `private Object readResolve() throws ObjectStreamException;`
Va trebui să returneze o copie a acestui obiect în scopuri de deserializare. Când o clasă care implementează această metodă este citită dintr-un `ObjectInputStream`, această metodă va fi apelată și va fi returnat obiectul rezultat. Aceasta va fi de obicei o instanță a unei clase care inițial furnizează o metodă `writeReplace()`.

5.4.4. Interfața Externalizable

Această interfață trebuie să fie implementată de clase care doresc să furnizeze facilități de serializare scrise de utilizator. Fluxurile de obiecte vor scrie doar numele clasei către flux; toate celelalte date trebuie comunicate prin această interfață. Clasele care utilizează acest mecanism nu câștigă nici o facilitate pusă la dispoziție de mecanismul de serializare implicit.

Un obiect `Externalizable` trebuie să furnizeze un constructor fără argumente prin care o instanță va fi creată anterior deserializării din flux. Metodele declarate în această interfață sunt:

- `void writeExternal(ObjectOutput ieseire) throws IOException;`
Va fi apelată când este serializată o instanță a acestei clase. Aceasta poate scrie toate datele către ieseire, folosind numeroasele metode declarate;
- `void readExternal(ObjectInput intrare) throws IOException;`
Va fi apelată când este deserializată o instanță a acestei clase. Aceasta poate citi toate datele din intrare, folosind numeroasele metode declarate. Crearea unei instanțe se va face de constructorul fără argumente, după care metoda va fi apelată.

Metodele de rezoluție și înlocuire pot fi de asemenea utilizate cu mecanismul de serializare `Externalizable`. O clasă `Externalizable` care dorește să pună la dispoziție rezoluția și înlocuirea poate implementa ambele metode `writeReplace()` și `readResolve()`, după care serializarea și deserializarea metodei corespunzătoare vor fi apelate automat.

5.4.5. Crearea unei clase serializabile

Implementarea unei clase serializabile ridică considerații de securitate, deoarece un obiect serializat își expune starea internă pentru citire și manipulare. Pot apărea și considerații de versiune, pentru că un obiect serializat poate mai târziu să fie citit înapoi într-un mediu cu o versiune mai veche sau mai nouă a fișierelor clasei actuale.

La implementarea unei clase trebuie ținut cont care părți ale stării clasei trebuie serializate. Implicit, toate datele nestatice și neocasionale (non-transient) vor fi serializate. Aceasta înseamnă că, dacă nu se dorește serializarea unor anumite variabile, atunci acestea trebuie declarate `transient`. Cu toate acestea, este bine să ne asigurăm în metodele `readObject()` și `writeObject()` că atributele `transient` vor fi corect inițializate după deserializare (altfel, acestea vor fi inițializate cu 0 sau `null` și ar conduce la o funcționare incorectă a programului).

JDK 1.2 introduce și un mecanism alternativ de declarare a câmpurilor care se doresc a fi serializate. Dacă o clasă declară un vector static din clasa `ObjectStreamField` numit `serialPersistentFields`, atunci doar câmpurile specificate în acest vector vor fi serializate. Acest mecanism se utilizează în aplicații care necesită serializare specială.

Există, în principiu, patru moduri standard diferite de creare a unei clase serializabile. Vom prezenta și varianta a cincea ca fiind cea mai generală modalitate de creare a unei clase serializabile.

5.4.5.1. Derivarea unei clase serializabile

Dacă derivăm o clasă care este deja serializată, avem prin tranzitivitate o subclasă serializată automat.

Exemplul 5.4.5. În programul Java de mai jos, clasa `UnApplet` este derivată din `Applet`, care la rândul ei este derivată din `Panel`, care este derivată din `Container`, care este derivată din `Component`. Cum clasa `Component` implementează interfața `Serializable`, rezultă că și clasa `UnApplet` este serializabilă. Fluxurile obiectului au grijă transparent de trimiterea tuturor stărilor appletului, inclusiv datele super-claselor, ca de altfel și noile date precum eticheta.

```
import java.awt.*;
import java.applet.*;

public class UnApplet extends Applet {
    private Label eticheta;

    public void init() {
        eticheta = new Label(getParameter("EtichetaUnu"));
        System.out.println(eticheta.getText());
    }
}
```

Pentru ca exemplul să funcționeze complet, în fișierul `.html` asociat poate fi scris:

```
<applet code="UnApplet" width="300" height="300">
    <param name="EtichetaUnu" value="PrimaEticheta">
</applet>
```

După apelarea acestui applet, în fereastra de ieșire (consolă) va fi afișat "Prima Eticheta".

5.4.5.2. Implementarea interfeței Serializable

Implementând interfața (de marcă) `Serializable`, noua clasă va prelua avantajele proprietăților de serializare a fluxurilor obiect.

Exemplul 5.4.6. Următorul program Java utilizează interfața `Serializable`:

```
import java.io.*;
import java.util.*;
```



```

class CercDoi implements Serializable {
    protected double x, y, raza;
    private Vector altCerc = new Vector();

    public CercDoi(double x, double y, double raza) {
        this.x = x;
        this.y = y;
        this.raza = raza;
    }

    public boolean punctInterior(double x, double y) {
        return (this.x - x) * (this.x - x) +
            (this.y - y) * (this.y - y) <= raza * raza;
    }

    public String toString() {
        return "Centrul (" + x + ", " + y + "), raza " + raza;
    }
}

public class TestCercDoiSerializare {
    public static void main(String args[]) {
        CercDoi obiect = new CercDoi(0, 0, 10);
        try {
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("cerc.dat"));
            oos.writeObject(obiect);
            oos.close();
        }
        catch (IOException e) {
            System.out.println("Eroare la serializare");
        }
        try {
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("cerc.dat"));
            CercDoi obiectCitit = (CercDoi) ois.readObject();
            System.out.println(obiectCitit);
            ois.close();
        }
        catch (IOException e) {
            System.out.println("Eroare la deserializare" +
                e.getMessage());
        }
    }
}

```

```

        catch (ClassNotFoundException e) {
            System.out.println("N-am gasit clasa obiectului in flux" +
                e.getMessage());
        }
    }
}

```

Cu toate că variabilele `x`, `y`, `raza` și `altCerc` nu sunt publice, acestea sunt accesibile fluxurilor obiect. Clasa `Vector` este serializabilă, deci toate elementele sale vor fi transmise cu succes. Metoda `readObject()` returnează un `Object` și deci trebuie realizat *downcasting* prin specificarea numelui clasei obiectului citit. La execuția programului de mai sus se va afișa:

```
Centrul (0.0, 0.0), raza 10.0
```

5.4.5.3. Metode de serializare obișnuite

Există multe cazuri când este util să dirijăm cumva procesul de serializare. În particular, este util când o clasă include date care nu sunt serializabile, care sunt unice pentru o mașină virtuală și trebuie transmise altele sau a căror transmisie nu este necesară.

Pentru a furniza serializare de această formă, o clasă trebuie să implementeze interfața `Serializable` și metodele `writeObject()` și `readObject()`. Spre deosebire de metodele cu același nume din clasele `ObjectOutputStream`, respectiv `ObjectInputStream`, metodele definite de noi nu au aceeași semnătură. Invers față de metodele din sistemul Java, acestea vor fi asociate obiectului curent al clasei care se serializează, iar ca parametru vor primi fluxul de obiect (intrare/ieșire). Acestea vor avea prototipul:

- `public void writeObject(ObjectOutputStream fluxObiectIesire) throws IOException;`
- `public void readObject(ObjectInputStream fluxObiectIntrare) throws IOException, ClassNotFoundException;`

Aceste metode trebuie declarate astfel încât să arunce orice tip de excepție. Cu toate acestea, programatorul trebuie să se asigure că se pot arunca doar aceste tipuri de excepții declarate de metodele corespunzătoare `writeObject()` și `readObject()`. De exemplu, metoda `writeObject()` poate arunca doar `IOException` sau o subclasă a acesteia, pe când `readObject()` poate arunca `IOException`, `ClassNotFoundException` sau o subclasă a acestora.

Exemplul 5.4.7. Fluxurile de obiecte nu pot transmite date transient, adică date care prin natura lor sunt unice. Datele transient (patratulRazei și `d`) vor fi procesate în metodele `writeObject()` și `readObject()`. Nici datele statice nu vor fi trimise de către fluxurile de obiecte. Datele statice sunt globale unei mașini virtuale și deci nu are sens transmiterea acestora printr-o singură instanță a clasei. Când se va primi fluxul de obiecte, datele statice (`globalId`) vor fi inițializate ca și cele normale.

```

import java.io.*;
import java.util.*;

```

```

class CercDoi implements Serializable {
    protected double x, y, raza;

    public CercDoi(double x, double y, double raza) {
        this.x = x;
        this.y = y;
        this.raza = raza;
    }

    public boolean punctInterior(double x, double y) {
        return (this.x - x) * (this.x - x) +
            (this.y - y) * (this.y - y) <= raza * raza;
    }

    public String toString() {
        return "Centrul (" + x + ", " + y + "), raza " + raza;
    }
}

class CercTrei extends CercDoi {
    private transient double patratulRazei;
    private transient int id;

    private static int globalId;
    private synchronized static int getId() {
        return globalId++;
    }

    public CercTrei() {super(0,0,0);}

    public CercTrei(double x, double y, double raza) throws
        IOException {
        super(x, y, raza);
        patratulRazei = raza * raza;
        id = getId();
    }

    public boolean punctInterior(double x, double y) {
        return (this.x - x) * (this.x - x) +
            (this.y - y) * (this.y - y) <= patratulRazei;
    }

    public void writeObject(ObjectOutputStream obiectIesire)
        throws IOException {
        obiectIesire.writeDouble(x);

```

```

        obiectIesire.writeDouble(y);
        obiectIesire.writeDouble(raza);
    }

    public void readObject(ObjectInputStream obiectIntrare)
        throws IOException, ClassNotFoundException {
        x = obiectIntrare.readDouble();
        y = obiectIntrare.readDouble();
        raza = obiectIntrare.readDouble();
        patratulRazei = raza * raza;
        id = getId();
    }

    public String toString() {
        return "Centrul (" + x + ", " + y + "), raza " + raza +
            " patratul Razei = " + patratulRazei + " id = " + id +
            " globalId = " + globalId;
    }
}

public class TestCercTreiSerializare {
    public static void main(String args[]) {
        CercTrei obiect = null;
        try {
            obiect = new CercTrei(1, 1, 10);
        }
        catch (IOException e) {
            System.out.println(
                "Eroare la crearea obiect din Clasa CercTrei");
        }
        System.out.println("obiect = " + obiect);
        try {
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("cerc.dat"));
            obiect.writeObject(oos);
            oos.close();
        }
        catch (IOException e) {
            System.out.println("Eroare la serializare " +
                e.getMessage());
        }
        try {
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("cerc.dat"));
            CercTrei obiectCitit = new CercTrei();

```

```

        obiectCitit.readObject(ois);
        System.out.println(obiectCitit);
        ois.close();
    }
    catch (IOException e) {
        System.out.println("Eroare la deserializare " +
            e.getMessage());
    }
    catch (ClassNotFoundException e) {
        System.out.println("N-am gasit clasa obiectului in flux" +
            e.getMessage());
    }
}
}

```

S-a marcat variabila `patratulRazei` cu `transient` pentru a face calculele din metoda `punctInterior()` mai eficient. Nu este necesară trimiterea acesteia, pentru că ea se poate refăce ca fiind pătratul razei. Variabila `id` a fost declarată `transient` pentru a defini un identificator unic pentru fiecare mașină virtuală. Fiecare instanță a clasei `CercTrei` dintr-o mașină virtuală va avea o valoare unică pentru `id`. Nu are sens să trimitem această dată, pentru că nu putem garanta că mașina virtuală primitoare nu a setat deja acest identificator.

Programul va afișa la execuție:

```

obiect = Centrul (1.0, 1.0), raza 10.0 patratul Razei = 100.0
id = 0 globalId = 1
Centrul (1.0, 1.0), raza 10.0 patratul Razei = 100.0 id = 1
globalId = 2

```

5.4.5.4. Implementarea interfeței `Externalizable`

Interfața `Externalizable` este un mecanism alternativ pentru ca o clasă să furnizeze serializabilitatea fără să țină cont de avantajele mecanismelor de serializare. Când este trimis către un flux un obiect `Externalizable`, se scrie un antet constând în principiu din numele clasei și atunci clasa este direct responsabilă de propria serializare și deserializare.

Exemplul 5.4.8. În acest exemplu, clasa `Dreptunghi` implementează interfața `Externalizable` și furnizează mecanisme manuale pentru serializare și deserializare. Pentru obiectele `Externalizable` trebuie să furnizăm un constructor public fără argumente care va crea inițial obiectul. Fluxurile obiect nu vor apela metodele unei superclase `Externalizable` (ele vor fi argumente), ci va trebui să apelăm explicit metodele superclasei asociate unui obiect din clasa `Dreptunghi` (care implementează `Externalizable`).

```

import java.io.*;

public class Extern {

```

```

public static void main(String args[]) {
    Dreptunghi d = new Dreptunghi(5, 20);
    try {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream("dreptunghi.dat"));
        d.writeExternal(oos);
        oos.close();
    }
    catch (IOException e) {
        System.out.println("Eroare " + e);
    }
    try {
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream("dreptunghi.dat"));
        Dreptunghi obiectCitit = new Dreptunghi();
        obiectCitit.readExternal(ois);
        System.out.println(obiectCitit);
        ois.close();
    }
    // la citire, pot interveni mai multe tipuri de exceptii !
    catch (IOException e) {
        System.out.println("Eroare la fisierul de intrare: " + e);
    }
    catch (ExceptionInInitializerError e) {
        System.out.println("Initializarea provocata de aceasta " +
            " metoda a esuat." + " Eroare: " + e);
    }
    catch (SecurityException e) {
        System.out.println("Nu avem permisiunea să cream o noua " +
            " instanta." + " Eroare: " + e);
    }
}

class Dreptunghi implements Externalizable {
    private int latime, lungime;

    public Dreptunghi (int latime, int lungime) {
        this.latime = latime;
        this.lungime = lungime;
    }

    public Dreptunghi() {}

    public void writeExternal(ObjectOutput obiectIesire)

```

```

throws IOException {
    obiectIesire.writeInt(latime);
    obiectIesire.writeInt(lungime);
}

public void readExternal(ObjectInput obiectIntrare)
throws IOException {
    latime = obiectIntrare.readInt();
    lungime = obiectIntrare.readInt();
}

public String toString() {
    return "Dreptunghi de latime " + latime + " si lungime "
        + lungime;
}
}

```

În urma execuției programul va afișa:

```
Dreptunghi de latime 5 si lungime 20
```

5.4.5.5. Declararea interfeței și a tuturor metodelor de acces

Se poate „cobori” și mai mult la nivelul serializării obiectelor și anume declarând interfața de serializare care va conține două metode abstracte:

```

public abstract void scrieDate(DataOutputStream dos);
public abstract void citeșteDate(DataInputStream dis);

```

Se utilizează clasa `DataOutputStream`, respectiv `DataInputStream`, pentru a scrie, respectiv a citi date de tipuri primitive către un, respectiv dintr-un flux. Către fluxul de ieșire se vor trimite numărul de obiecte și numele clasei, urmat de obiectele respective. Acestea vor fi preluate din fluxul de intrare în aceeași ordine, adică numărul de obiecte, numele clasei și apoi obiectele. Pentru crearea unui obiect de un anumit tip de clasă, apelăm metoda statică `forName()` din clasa `Class`. Sub JDK 1.1 există:

```

public static Class forName(String className) throws
    ClassNotFoundException;

```

iar sub JDK 1.2, există și o altă variantă:

```

public static Class forName(String nume, boolean initializator,
    ClassLoader incarcator) throws ClassNotFoundException;

```

care returnează obiectul `Class` asociat cu nume, folosind valoarea variabilei `incarcator`. Clasa este inițializată doar dacă valoarea lui `initializator` este `true`.

Apoi, putem utiliza metoda `newInstance()` din clasa `Class` pentru a crea un obiect al unei clase fără a apela operatorul `new`. Sintaxa este:

```

public Object newInstance() throws InstantiationException,
    IllegalAccessException;

```

Exemplul 5.4.9. Se vor defini 10 obiecte serializabile, din care primele 5 vor aparține clasei `PrimaClasa`, iar următoarele 5 clasei `ADouaClasa`.

```

import java.io.*;

public class SerializareObiectePrimitive {
    public static void main(String args[]) {
        // declarăm 10 obiecte pe care le vom serializa
        Serializare A[] = new Serializare[10];
        int i;
        // vom defini 5 obiecte din PrimaClasa si ultimele 5 din
        // ADouaClasa
        for (i = 0; i < A.length/2; i++)
            A[i] = new PrimaClasa(i, (double) i * 1.5, "test" + i,
                i * 2);
        for (i = A.length/2; i < A.length; i++)
            A[i] = new ADouaClasa(i * 3);
        /* vom declara un fisier in acces de scriere. Acesta va
        avea structura: numar de obiecte, apoi reprezentarea
        obiectelor (numele clasei urmat de obiectul
        respectiv) */
        try {
            FileOutputStream fos = new FileOutputStream(
                "Fisier1.tmp");
            DataOutputStream dos = new DataOutputStream(fos);
            dos.writeInt(A.length);
            for (i = 0; i < A.length; i++) {
                dos.writeUTF(A[i].getClass().getName());
                A[i].scrieDate(dos);
            }
            fos.close();
        }
        catch (IOException e) {
            System.out.println("Eroare " + e);
        }
        // declarăm un alt vector din interfata Serializare
        Serializare B[] = null;
        // vom declara un fisier in acces de citire. Acesta va fi
        // citit dupa structura: numar de obiecte, apoi reprezentarea
        // obiectelor (numele clasei urmat de obiectul respectiv)
        try {
            FileInputStream fis = new FileInputStream(
                "Fisier1.tmp");
            DataInputStream dis = new DataInputStream(fis);

```

```

    int numarObiecte = dis.readInt();
    B = new Serializare[numarObiecte];
    for (i = 0; i < B.length; i++) {
        Class C = Class.forName(dis.readUTF());
        B[i] = (Serializare) C.newInstance();
        B[i].citesteDate(dis);
        System.out.println(B[i]);
    }
    dis.close();
}
// la citire, pot interveni mai multe tipuri de exceptii !
catch (IOException e) {
    System.out.println("Eroare la fisierul de intrare: " + e);
}
catch (InstantiationException e) {
    System.out.println(
        "Clasa C este abstracta, interfața, sir, "
        + "tip primitiv sau void. " + "Eroare: " + e);
}
catch (IllegalAccessException e) {
    System.out.println("Clasa sau initializatorul " +
        "nu este accesibil. Eroare " + e);
}
catch (ClassNotFoundException e) {
    System.out.println("Nu am gasit clasa cu numele " +
        "respectiv. Eroare " + e);
}
catch (ExceptionInInitializerError e) {
    System.out.println("Initializarea provocata " +
        "de aceasta metoda a esuat. Eroare: " + e);
}
catch (SecurityException e) {
    System.out.println("Nu avem permisiunea sa cream " +
        "o noua instanta. Eroare: " + e);
}
}

class PrimaClasa implements Serializare {
    int intreg = 0;
    double dublu = 0.0;
    String sir = null;
    ADouaClasa obiectDoi = null;

```

```

    public PrimaClasa() {}
    public PrimaClasa(int i, double d, String s, int numarDoi) {
        intreg = i;
        dublu = d;
        sir = s;
        obiectDoi = new ADouaClasa(numarDoi);
    }

    public void scrieDate(DataOutputStream dos) {
        try {
            dos.writeInt(intreg);
            dos.writeDouble(dublu);
            dos.writeUTF(sir);
            obiectDoi.scrieDate(dos);
        }
        catch (IOException e) {
            System.out.println("Eroare: " + e);
        }
    }

    public void citesteDate(DataInputStream dis) {
        try {
            intreg = dis.readInt();
            dublu = dis.readDouble();
            sir = dis.readUTF();
            obiectDoi = new ADouaClasa();
            obiectDoi.citesteDate(dis);
        }
        catch (IOException e) {
            System.out.println("Eroare: " + e);
        }
    }

    public String toString() {
        /* in cazul lui obiectDoi se apeleaza metoda toString() din
           clasa ADouaClasa */
        return "intreg = " + intreg + ", dublu = " + dublu +
            ", sir = " + sir + ", obiectDoi: " + obiectDoi;
    }
}

class ADouaClasa implements Serializare {
    int intreg = 0;

```

```

public ADouaClasa() {}

public ADouaClasa(int i) {
    intreg = i;
}

public void scrieDate(DataOutputStream dos) {
    try {
        dos.writeInt(intreg);
    }
    catch (IOException e) {
        System.out.println("Eroare: " + e);
    }
}

public void citesteDate(DataInputStream dis) {
    try {
        intreg = dis.readInt();
    }
    catch (IOException e) {
        System.out.println("Eroare: " + e);
    }
}

public String toString() {
    return "intreg = " + intreg;
}

interface Serializare {
    public abstract void scrieDate(DataOutputStream dos);
    public abstract void citesteDate(DataInputStream dis);
}

```

Programul va afișa la execuție:

```

intreg = 0, dublu = 0.0, sir = test0, obiectDoi: intreg = 0
intreg = 1, dublu = 1.5, sir = test1, obiectDoi: intreg = 2
intreg = 2, dublu = 3.0, sir = test2, obiectDoi: intreg = 4
intreg = 3, dublu = 4.5, sir = test3, obiectDoi: intreg = 6
intreg = 4, dublu = 6.0, sir = test4, obiectDoi: intreg = 8
intreg = 15
intreg = 18
intreg = 21

```

```

intreg = 24
intreg = 27

```

5.5. Concluzii

Capitolul de față se referă la fișiere de intrare/ieșire (Secțiunile 5.3.1 și 5.3.2) și la fluxuri de intrare/ieșire (Secțiunea 5.3.3). Există fluxuri de intrare/ieșire de nivel jos (Subsecțiunea 5.3.3.1), fluxuri de intrare/ieșire filtru de nivel înalt (Subsecțiunea 5.3.3.2) și fluxuri de citire/scriere (Subsecțiunea 5.3.3.3). Secțiunea 5.4 descrie procedeele de serializare în Java.

5.6. Test grilă

Întrebarea 5.6.1. Care dintre afirmațiile de mai jos sunt adevărate?

- a) caracterele UTF se reprezintă pe 8 biți;
- b) caracterele UTF se reprezintă pe 16 biți;
- c) caracterele UTF se reprezintă pe 24 biți;
- d) caracterele Unicode se reprezintă pe 16 biți;
- e) caracterele Bytecode se reprezintă pe 16 biți.

Întrebarea 5.6.2. Care din afirmațiile de mai jos sunt adevărate?

- a) când se creează o instanță a lui `File` și nu se folosește semantica de denumire a fișierelor de pe mașina locală, atunci constructorul va arunca o excepție `IOException`;
- b) când se creează o instanță a lui `File` și fișierul respectiv nu există pe sistemul de fișiere local, atunci se creează un astfel de fișier;
- c) când o instanță a lui `File` este eliberată de *garbage collector*, atunci fișierul corespunzător de pe sistemul local este șters;
- d) nici una dintre afirmațiile de mai sus nu este adevărată.

Întrebarea 5.6.3. Pentru afirmațiile de mai jos, selectați pe cele adevărate:

- Clasa `File` conține o metodă care schimbă directorul de lucru curent;
- Clasa `File` conține o metodă care listează conținutul directorului de lucru curent.
- a) Nu, Nu
- b) Da, Nu
- c) Nu, Da
- d) Da, Da

Întrebarea 5.6.4. Câți octeți va scrie codul Java de mai jos către fișierul `fisier.txt`?

```

try {
    FileOutputStream f1 = new FileOutputStream("fisier.txt");
    DataOutputStream f2 = new DataOutputStream(f1);

```

```
f2.writeInt(3);
f2.writeDouble(0.01);
f2.close();
f1.close();
}
```

- a) 2;
- b) 8;
- c) 12;
- d) 16;
- e) numărul de octeți va depinde de sistemul local respectiv.

Întrebarea 5.6.5. Ce va afișa subcodul Java de mai jos la linia 8?

```
1. FileOutputStream f1 = new FileOutputStream("fisier.txt");
2. for (byte b = 10; b < 50; b++) f1.write(b);
3. f1.close();
4. RandomAccessFile f2 = new RandomAccessFile("fisier.txt", "r");
5. f2.seek(10);
6. int i = f2.read();
7. f2.close();
8. System.out.println("i = " + i);
```

- a) i = 10
- b) i = 20
- c) i = 30
- d) se va arunca o excepție la linia 1;
- e) se va arunca o excepție la linia 4.

Întrebarea 5.6.6. Presupunem că utilizăm codul de mai jos pentru a crea un fișier:

```
FileOutputStream f1 = new FileOutputStream("fisier.txt");
DataOutputStream f2 = new DataOutputStream(f1);
for (int i = 0; i < 500; i++)
    f2.writeInt(i);
```

Pentru a citi date din acest fișier, care dintre soluțiile de mai jos este corectă?

- a) Construim un FileInputStream căruia i se trimite ca parametru numele fișierului. Folosind acest obiect, se creează un obiect DataInputStream căruia i se apelează metoda readInt();
- b) Construim un FileReader căruia i se trimite ca parametru numele fișierului. Folosind acest obiect, se apelează metoda readInt();
- c) Construim un PipedInputStream căruia i se trimite ca parametru numele fișierului. Folosind acest obiect, se apelează metoda readInt();
- d) Construim un RandomAccessFile căruia i se trimite ca parametru numele fișierului. Folosind acest obiect, se apelează metoda readInteger();

- e) Construim un FileReader căruia i se trimite ca parametru numele fișierului. Folosind acest obiect, se creează un obiect DataInputStream căruia i se apelează metoda readInt().

Întrebarea 5.6.7. Pentru afirmațiile de mai jos, selectați pe cele adevărate:

Clasa Readers conține o metodă care citește și returnează o valoare de tip float;
Clasa Writers conține o metodă care scrie o valoare de tip float.

- a) Nu, Nu
- b) Da, Nu
- c) Nu, Da
- d) Da, Da

Întrebarea 5.6.8. Fie codul Java de mai jos. Care este rezultatul execuției lui?

```
1. File f1 = new File("numeDirector");
2. File f2 = new File(f1, "numeFisier");
```

- a) eroare de compilare la linia 2, deoarece avem un apel recursiv nepermis;
- b) programul este corect, dar nu se creează nici un director și nici un fișier;
- c) programul este corect și se creează în directorul curent un director cu numele numeDirector, iar în acest director se creează fișierul cu numele numeFisier;
- d) programul este corect și se creează în directorul curent un director cu numele numeDirector și fișierul cu numele numeFisier;
- e) programul este corect și se creează în directorul curent doar fișierul cu numele numeFisier.

Întrebarea 5.6.9. Presupunem că avem un fragment de cod Java dintr-o aplicație care are drept de scriere în directorul de lucru curent și că în directorul curent nu există fișierul numit fisier.txt. Care va fi rezultatul compilării și execuției codului de mai jos?

```
1. try {
2.     RandomAccessFile f1 = new RandomAccessFile(
        "fisier.txt", "rw");
3.     BufferedOutputStream f2 = new BufferedOutputStream(f1);
4.     DataOutputStream f3 = new DataOutputStream(f2);
5.     f3.writeDouble(Math.PI);
6.     f3.close();
7.     f2.close();
8.     f1.close();
9. } catch (IOException e) { }
```

- a) Codul Java de mai sus nu se poate compila;
- b) Codul Java de mai sus se poate compila, dar aruncă o excepție la linia 3;
- c) Codul Java de mai sus se poate compila și executa, dar nu are nici un efect pe sistemul de fișiere local;

- d) Codul Java de mai sus se poate compila și executa, creându-se fișierul `fișier.txt` în directorul curent.

Întrebarea 5.6.10. Presupunem că avem o clasă `C1` care extinde interfața `Serializable`. Selectați o opțiune de mai jos în funcție de valoarea de adevăr a următoarelor afirmații:

1. Întotdeauna clasa `C1` trebuie să redefinească metodele `readObject()` și `writeObject()`;
2. Dacă `C1` extinde clasa `Applet`, atunci nu este nevoie să se redefinească metodele `readObject()` și `writeObject()`;
3. Atributele `transient` din clasa `C1` nu vor fi serializate.
 - a) 1, 2, 3
 - b) 1, 2
 - c) 1, 3
 - d) 2, 3
 - e) 2

Întrebarea 5.6.11. Presupunem că avem un program Java și că în directorul de lucru curent există drepturi de scriere. Care va fi rezultatul compilării și execuției codului de mai jos?

```
import java.io.*;

class C1 {
    private double a = 3.45;
    public String toString() { return "a = " + a; }
}

public class TestGrilaSerializare {
    public static void main(String args[]) {
        C1 obiect = new C1();
        try {
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("intrare.dat"));
            oos.writeObject(obiect);
            oos.close();
        }
        catch (IOException e) {
            System.out.println("Eroare la serializare " +
                e.getMessage());
        }
        try {
            ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("intrare.dat"));
            C1 obiectCitit = (C1) ois.readObject();
            System.out.println(obiectCitit);
            ois.close();
        }
```

- ```

 }
 catch (IOException e) {
 System.out.println("Eroare la deserializare " +
 e.getMessage());
 }
 catch (ClassNotFoundException e) {
 System.out.println("N-am gasit clasa obiectului in flux" +
 e.getMessage());
 }
}
}

```
- a) Codul Java de mai sus nu se poate compila;
  - b) Codul Java de mai sus se poate compila, dar aruncă o excepție la serializare;
  - c) Codul Java de mai sus se poate compila, dar aruncă o excepție la deserializare;
  - d) Codul Java de mai sus se poate compila, dar aruncă două excepții (la serializare și la deserializare);
  - e) Codul Java de mai sus se poate compila și executa, afișându-se textul `a = 3.45`.

## 5.7. Exerciții propuse spre implementare

**Exercițiul 5.7.1.** Scrieți un program Java care modifică programul `Gaseste.java` (Exemplul 5.3.11) astfel încât să afișezi și numerele liniilor care conțin acel cuvânt. În final, afișezi și numărul total de apariții ale cuvântului căutat. Toate aceste informații trebuie scrise într-un fișier extern. În cazul în care cuvântul nu este găsit în fișier, afișezi un mesaj corespunzător.

**Exercițiul 5.7.2.** Scrieți o aplicație Java care listează pe ecran conținutul unui director împreună cu precizarea faptului dacă este subdirector sau fișier, furnizând și lungimile (în octeți) ale fișierului respectiv. Numele directorului al cărui conținut trebuie listat se dă ca parametru din linia de comandă. Furnizați numărul total de octeți ocupați de director.

**Exercițiul 5.7.3.** Scrieți un program Java care citește un fișier text și creează un alt fișier pornind de la primul în care liniile sunt numerotate.

**Exercițiul 5.7.4.** Utilizând clasa `FileRandomAccess`, scrieți un program Java care primește la intrare un fișier binar de numere întregi (sau de tip `float`, `double` etc.) și listează fiecare element (la ecran și în acel fișier binar) înlocuind numerele negative cu modulul acestora.

**Exercițiul 5.7.5.** Scrieți două funcții Java care întorc `true`, dacă pointerul fișierului vizează sfârșitul fluxului, respectiv al fișierului cu acces aleator. De exemplu, acestea pot avea prototipurile:



```
public static boolean eof(InputStream fisier);
public static boolean eof(RandomAccessFile fisier);
```

**Exercițiul 5.7.6.** Fie un tablou de 8 elemente de tip `int` și altul de 12 elemente de tip `double`. Să se serializeze și deserializeze acești vectori. Folosiți toate cele 5 metode de serializare/deserializare. Care este cea mai avantajoasă?

**Exercițiul 5.7.7.** Presupunem că avem un tablou de dimensiune foarte mare care nu poate fi alocat contiguu în memorie. Să se împartă în subtablouri mai mici cărora li se poate alocă spațiu de memorie care apoi se serializează în fișiere de pe disc. Când se dorește reutilizarea lor, acestea se deserializează de pe disc.

## 5.8. Proiecte propuse spre implementare

**Proiectul 5.8.1.** (Pretty printing) Scrieți un program Java care primește la intrare un program Java (eventual „scris neuniform”) și îl transformă într-un program Java scris respectând regulile stilului profesionist de redactare a programelor (indentare a sub-instrucțiunilor, o singură instrucțiune pe rând, fără mai mult de o linie vidă etc.).

## 6. Fire de execuție

### 6.1. Cuvinte cheie

- fire de execuție, sincronizare și paralelism
- excludere mutuală, prioritate
- clasa `Thread` și interfața `Runnable`
- grupuri de fire de execuție

## 6.2. Introducere

### 6.2.1. Breviar teoretic

*Program concurrent* = mulțime de programe secvențiale (task-uri, procese) care se execută într-un cadru paralel (abstract). Cadrul paralel (abstract) nu va impune ca fiecare proces să se execute pe un procesor fizic separat.

Arhitecturile de calculatoare utilizate pentru execuția programelor concurente sunt:

- memorie comună = computere în care procesele își partajează accesul la locațiile de memorie sau sunt capabile să apeleze un sistem de operare centralizat pentru a primi un serviciu. Algoritmii de tip memorie comună corespund programelor concurente care se execută prin partajarea unui singur computer sau arhitecturilor multiprocesor care partajează o memorie;
- sisteme distribuite = computere care comunică prin schimb de mesaje.

*Programare concurentă abstractă* = studiul execuției intercalate (eng. *interleaving*) a secvențelor de instrucțiuni atomice ale mai multor procese secvențiale. Faptul că este abstract înseamnă că avem un model idealizat al unui fenomen care ignoră detaliile instrucțiunilor mașină specifice sau detaliile de arhitectură.

Tehnici de abstractizare utilizate în crearea sistemelor software:

- *încapsulare* = punerea elementelor de date și a procedurilor ce operează asupra lor într-un singur modul;
- *concurența* = abstractizarea necesară studiului comportării dinamice a programelor.

Secțiunea critică reprezintă o resursă a unui calculator (memorie, imprimanta etc.) la care accesul se realizează secvențial (la un moment dat, un singur proces are acces la respectiva resursă).

### 6.2.2. Corectitudinea programelor

Există două definiții pentru corectitudinea programelor secvențiale care se presupun că se termină. Fie  $P(x)$  proprietatea variabilelor de intrare  $x$  și  $Q(x, y)$  proprietatea variabilelor de intrare  $x$  și variabilelor de ieșire  $y$ .

Corectitudine parțială:  $(P(a) \text{ și } \text{termină}(\text{Prog}(a, b))) > Q(a, b)$

Corectitudine totală:  $P(a) > (\text{termină}(\text{Prog}(a, b)) \text{ și } Q(a, b))$

Un program care intră totdeauna într-o buclă infinită este parțial corect pentru orice specificare.

Există două tipuri de proprietăți de corectitudine pentru programele concurente:

- Safety properties* (proprietatea de siguranță): proprietatea trebuie să fie totdeauna adevărată:
  - excludere mutuală: în secțiunea critică există (în orice moment) cel mult un proces;

- absența deadlock-ului: dacă există  $N$  ( $N \geq 2$ ) procese care doresc să acceseze secțiunea critică, atunci există unul care reușește.

- Liveness properties* (proprietatea de continuitate): proprietatea trebuie să fie eventual adevărată:

- absența inaniției (eng. *starvation*): dacă un proces își anunță intenția de a intra în secțiunea critică, atunci el are șanse reale de reușită (nu trebuie să avem procese defavorizate);
- accesarea secțiunii critice în absența disputei (eng. *contention*): dacă numai un proces dorește să acceseze secțiunea critică, atunci el trebuie să reușească.

În cazul disputelor pentru secțiunea critică, se cunosc mai multe moduri de rezolvare (proprietăți fairness), din care două pot fi implementate ușor:

- **Linear waiting**: dacă un proces face o cerere, el va fi servit înainte ca orice alt proces să fie servit de două ori;
- **FIFO waiting**: dacă un proces  $P$  face o cerere, el va fi servit înaintea oricărui proces ce face ulterior aceeași cerere (First In, First Out).

Dacă se cunoaște timpul maxim permis unui proces să rezolve o cerere, se va putea estima perioada de timp după care va fi servită o cerere anume. Soluția FIFO waiting este uzuală în sistemele centralizate.

### 6.2.3. Primitivele de programare concurentă

Problema excluderii mutuale pentru  $N$  procese ( $N = 2$ ) se poate rezolva apelând la:

- instrucțiuni atomice (LOAD, STORE etc.). Se cunosc algoritmii Dekker, Petterson;
- primitive de programare concurentă (semafoare, monitoare).

*Monitorul* este primitivă de programare concurentă structurată, care concentrează răspunderea corectitudinii numai în câteva module.

Etimologie: a monitoriza, a superviza.

Modul de funcționare este următorul: pentru fiecare obiect sau grup de obiecte (de exemplu, alocarea dispozitivelor de intrare/ieșire sau a memoriei, cererile de intrare/ieșire) se definește câte un monitor. Procesele cer servicii de la diferitele monitoare. Dacă același monitor este apelat de două procese, implementarea asigură că acestea sunt procesate secvențial pentru a asigura excluderea mutuală. Dacă sunt apelate mai multe monitoare, execuțiile lor se pot intercala.

Sintaxa monitoarelor se bazează pe încapsularea elementelor de date și a procedurilor care operează asupra lor într-un singur modul. Interfața cu un monitor constă într-o mulțime de proceduri care acționează asupra unor date ascunse în modul. Monitorul protejează datele interne de accese nerestricționate și în plus sincronizează apelurile la procedurile de interfață (spre deosebire de un modul obișnuit).

Pentru sincronizare definim o variabilă de condiție  $C$  care are asociate trei operații:

- **Wait( $C$ )** - procesul care cheamă procedura monitor ce conține această instrucțiune este suspendat într-o coadă FIFO asociată cu  $C$ ;

- Signal(C) - dacă coada pentru C este nevidă, atunci activează procesul din capul cozii;
- Non-empty(C) - returnează adevărat, dacă coada pentru C este nevidă.

### 6.3. Programare concurentă în Java

Programarea cu fire de execuție (eng. *multithread*) este un aspect important al limbajului Java. Sinonime pentru multithreading există în literatura de specialitate: paralelism și concurență.

Multitasking înseamnă capacitatea unui computer de a executa mai multe programe (procese, task-uri) în același timp (de exemplu, pe sistemele de operare de tip Windows (95, 98, NT, 2000, XP), Linux ș.a.). Multitasking nu implică neapărat un sistem multiprocesor, ci poate avea loc și pe un singur procesor. Sistemul de operare (la sistemele monoprocessor) alocă câte o cantitate de timp pentru execuția unui task. Task-urile care nu se execută își așteaptă rândul într-o coadă de așteptare (create de sistemul de operare pe baza unui mecanism de priorități).

Multithreading înseamnă capacitatea unui program de a executa mai multe secvențe de cod în același timp. O astfel de secvență de cod se numește fir de execuție sau thread. Datorită posibilității creării mai multor thread-uri, un program Java poate să execute mai multe sarcini simultan (de exemplu, animația unei imagini, transmiterea unei melodii la placa de sunet, comunicarea cu un server).

Limbajul Java suportă multithreading prin clase disponibile în pachetul `java.lang`, care este pachetul fundamental al limbajului Java.

În pachetul `java.lang` există două clase și o interfață cu care se pot dezvolta programe multithread: clasele `Thread` și `ThreadGroup` și interfața `Runnable`.

Clasa `Thread` și interfața `Runnable` oferă suport pentru lucrul cu thread-uri ca entități separate, iar clasa `ThreadGroup`, pentru crearea unor grupuri de thread-uri în vederea tratării acestora într-un mod unitar.

Clasa `Thread` implementează interfața `Runnable`, iar obiectele de tip `ThreadGroup` conțin mai multe obiecte de tip `Thread`. Un `ThreadGroup` poate conține mai multe `ThreadGroup`-uri.

Există două metode pentru crearea unui thread:

- creăm o clasă derivată din clasa `Thread`;
- creăm o clasă care implementează interfața `Runnable`.

#### 6.3.1. Crearea unui fir de execuție prin extinderea clasei `Thread`

Pentru ca firul de execuție să ruleze propria metoda `run()`, trebuie să extindem clasa `Thread` și să redefinim metoda `run()`. Există următoarele etape:

- crearea unei clase derivate din clasa `Thread` (extends `Thread`);
- suprascrierea funcției `public void run()` moștenită din clasa `Thread`. De exemplu, metoda `main()` este apelată de Java Runtime System. Metoda `run()` este metoda apelată când se execută un thread. Când se pornește JVM, se

pornește un thread care apelează metoda `main()`. JVM rulează atât timp cât toate thread-urile sunt în execuție și nu se apelează metoda `exit()` din clasa `System`;

- instanțierea unui obiect *thread* folosind `new`;
- pornirea thread-ului instanțiat, prin apelul metodei `start()` moștenită din clasa `Thread`. Apelul acestei metode face ca mașina virtuală Java să creeze contextul de program necesar unui thread după care să apeleze metoda `public void run()`.

Exemplul 6.3.1. Iată un prim program demonstrativ foarte simplu:

```
public class PrimulFir {
 public static void main(String args[]) {
 System.out.println("Crearea firului de execuție");
 FirExecutie fir = new FirExecutie();
 System.out.println("Startam firul de execuție");
 fir.start();
 System.out.println("Revenim in main()");
 }
}

class FirExecutie extends Thread {
 public void run() {
 int numarPasi = 5;
 System.out.println("Run se executa de " + numarPasi +
 " ori.");
 for (int i = 1; i <= numarPasi; i++)
 System.out.println("Pasul " + i);
 System.out.println("Run si-a terminat treaba");
 }
}
```

Mesajul "Revenim in main()" apare înaintea mesajului "Run se executa de 5 ori". Metoda `main()` are propriul său thread în care se execută. Prin apelul `start()` se cere mașinii virtuale Java crearea și pornirea unui nou thread. Din funcția `start()` se va ieși imediat. Thread-ul corespunzător metodei `main()` își continuă execuția independent de thread-ul proaspăt creat.

Exemplul 6.3.2. Putem crea mai multe fire de execuție, ba chiar și cu nume.

```
public class AlDoileaFir {
 public static void main(String args[]) {
 FirExecutie fir1 = new FirExecutie("FIR1");
 FirExecutie fir2 = new FirExecutie("FIR2");
 System.out.println(
 "JVM va crea contextul celor doua fire de execuție!");
 fir1.start();
 }
}
```

```

 fir2.start();
 System.out.println(
 "JVM a creat contextul celor doua fire de executie!");
 }
}

class FirExecutie extends Thread {
 private String numeFir;

 public FirExecutie(String nume) {
 numeFir = nume;
 }

 public void run() {
 int numarPasi = 5;
 System.out.println("Run se executa de " + numarPasi +
 "ori.");
 for (int i = 1; i <= numarPasi; i++) {
 System.out.println(numeFir + " este la pasul " + i);
 try {
 sleep(500);
 }
 catch (InterruptedException e) {
 System.err.println("Eroare in somn");
 }
 }
 System.out.println(numeFir + " si-a terminat treaba");
 }
}

```

Observăm că cele două fire de execuție rulează concomitent.

### 6.3.2. Crearea unui thread utilizând interfața Runnable

Această modalitate este de foarte mare ajutor, deoarece astfel clasa de tip Thread pe care o implementăm poate moșteni capabilități de la altă clasă (Java nu permite moștenirea multiplă cu extends!). Firul de execuție va rula metoda run() pentru alte obiecte. Pentru aceasta se utilizează o altă formă de constructor Thread:

```

public Thread(Runnable destinatie);

```

Interfața Runnable descrie o singură metodă, și anume run(). Operațiile de creare a unui thread folosind interfața Runnable sunt:

- crearea unei clase care să implementeze interfața Runnable; de exemplu:

```

class FirExecutie extends Afisare implements Runnable { ... }

```

- implementarea metodei run() din interfața Runnable;
  - instanțierea unui obiect al clasei utilizând new:
- ```

FirExecutie firExecutie = new FirExecutie();

```
- crearea unui obiect din clasa Thread folosind un constructor care are ca parametru un obiect de tip Runnable:
- ```

Thread fir = new Thread(firExecutie);

```
- se pornește firul de execuție (fir):
- ```

fir.start();

```

Exemplul 6.3.3. Un exemplu simplu de utilizare a interfeței Runnable:

```

/** clasa principala */
public class PrimulRunnable {
    public static void main(String args[]) {
        System.out.println("Crearea obiectului runnable");
        Tinta tinta = new Tinta();
        System.out.println("Cream firul de executie");
        Thread fir = new Thread(tinta);
        System.out.println("Startam firul de executie");
        fir.start();
        System.out.println("Revenim in main()");
    }
}

class Afisare {
    public void afisare(String mesaj) {
        System.out.println(mesaj);
    }
}

/** clasa corespunzatoare firelor de executie */
class Tinta extends Afisare implements Runnable {
    public void run() {
        int numarPasi = 5;
        afisare("Run se executa de " + numarPasi + " ori.");
        for (int i = 1; i <= numarPasi; i++)
            afisare("Pasul " + i);
        afisare("Run si-a terminat treaba");
    }
}

```

În urma execuției programului de mai sus vom obține următorul rezultat:

```

Crearea obiectului runnable
Cream firul de executie

```

```

Startam firul de executie
Revenim in main()
Run se executa de 5 ori.
Pasul 1
Pasul 2
Pasul 3
Pasul 4
Pasul 5
Run si-a terminat treaba

```

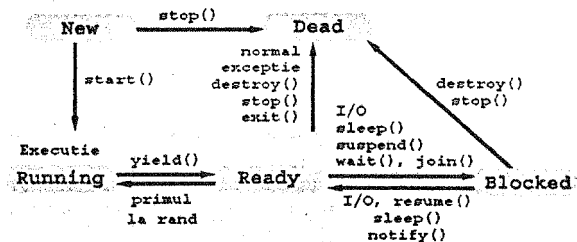
Se observă faptul că imediat după pornirea firului de execuție metoda `main()` se termină, iar firul de execuție își continuă execuția.

6.3.3. Controlul unui fir de execuție

Un fir de execuție se poate afla la un moment dat în una din următoarele stări: *running* (starea spre care aspiră toate firele de execuție), *waiting* (starea de așteptare, adormire, suspendare, blocare), *ready* (gata de execuție, prezent în coada de așteptare), *dead* (terminat). Fiecare thread are o prioritate de execuție. În general, thread-ul cu prioritatea cea mai mare este cel care va accesa primul resursele sistem. O clasă aparte de thread-uri sunt cele de tip *Daemon* care sunt thread-uri de serviciu (aflate în serviciul altor thread-uri).

Când se pornește mașina virtuală Java, există un singur thread care nu este de tip *Daemon* și care de obicei apelează metoda `main()`. JVM rămâne pornită atâta vreme cât există activ un thread care să nu fie de tipul *Daemon*.

Iată un automat finit determinist care precizează starea unui fir de execuție, precum și cum trecem dintr-o stare în alta:



Un thread instanțiat cu `new` se află într-o stare inițială. El poate trece în starea de execuție apelând `start()` sau se poate termina apelând `stop()` (de fapt, această metodă este învechită – *deprecated*). Starea de execuție are de fapt două substări: execuția propriu-zisă și așteptarea în coadă. JVM alocă fiecărui thread câte o cantitate de timp pentru execuție, făcând execuția aparent simultană a mai multor fire de execuție.

După ce un thread își epuizează timpul alocat, se întrerupe execuția acestuia și este așezat în coada de așteptare, cedându-se locul altui thread. Când îi vine rândul,

thread-ul pus în coadă își va relua execuția din punctul în care a fost oprit. Un fir de execuție poate fi forțat să fie pus în coadă folosind funcția `yield()`. Din starea de execuție, thread-ul poate trece în starea *dead* fie prin terminarea normală a metodei `run()`, fie prin aruncarea unei excepții, fie prin apelarea metodei `stop()` sau fie prin stoparea JVM (`exit()`).

Din starea de execuție, thread-ul poate trece într-o stare de blocare temporară. Unul dintre aceste cazuri este apelul `suspend()`, revenirea realizându-se cu apelul lui `resume()` (de fapt, metodele `suspend()` și `resume()` sunt acum învechite). Tot în starea de blocare, thread-ul poate trece prin apelul metodei `sleep()`. Se revine înapoi când se termină timpul dat prin parametrii metodei sau în momentul în care se aruncă o excepție de tipul `InterruptedException` prin apelul lui `interrupt()`.

Tot în starea blocat se poate ajunge și prin apelul metodei `join()`, caz în care revenirea se face în momentul când thread-ul, după ce așteaptă, și-a terminat execuția. Cazul `wait()` - `notify()` se referă la sincronizarea thread-urilor (vom reveni). Blocarea are loc în cazul unei operații de I/O. Thread-ul rămâne blocat până la terminarea operației. Tot în această stare se poate trece printr-un apel al lui `synchronized`. Din această stare de blocare temporară se trece în starea *dead* apelând `stop()`. Metoda `destroy()` distruge thread-ul fără a face dealocările necesare.

Putem afla starea unui thread apelând funcția:

```
public final native boolean isActive();
```

Cuvântul rezervat `native` precizează că funcția este implementată într-o bibliotecă externă.

Aceasta returnează `true`, dacă thread-ul este în execuție, sau blocat și `false`, dacă thread-ul este nou creat sau terminat (*dead*). Un fir de execuție aflat în starea *dead* nu mai poate fi repornit. Această încercare este privită ca o eroare de execuție furnizând excepția:

```
java.lang.IllegalThreadStateException
```

Exemplul 6.3.4. Prezentăm o mică generalizare a problemei cu mingea care sare într-un dreptunghi. Programul poate starta mai multe mingi, pe care apoi le oprim (folosim un vector de cel mult 10 fire de execuție).

```

import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

```

```

public class Bouncer extends Applet implements ActionListener {
    private Button start, stop, stare;
    private Ball firExecutie[] = new Ball[10];
    private static int numarFirExecutie = -1;
    private int count = 0;

    public void init() {

```

```

start = new Button("Start (cel mult 10 mingi)");
add(start);
start.addActionListener(this);
stop = new Button("Stop");
add(stop);
stop.addActionListener(this);
stare = new Button("Stare");
add(stare);
stare.addActionListener(this);
}

public void actionPerformed(ActionEvent event) {
    if (event.getSource() == start)
        if (numarFirExecutie < 9) {
            Graphics g = getGraphics();
            numarFirExecutie++;
            firExecutie[numarFirExecutie] = new Ball(g);
            firExecutie[numarFirExecutie].start();
        }
    if (event.getSource() == stop)
        if (numarFirExecutie >= 0)
            firExecutie[numarFirExecutie--].pleaseStop();
    if (event.getSource() == stare && numarFirExecutie >= 0) {
        count++;
        Graphics g = getGraphics();
        if (firExecutie[numarFirExecutie].isAlive())
            g.drawString("The ball " + (numarFirExecutie + 1) +
                " is alive (running or blocked)", 20, 135 + count
                * 15);
        else
            g.drawString("The ball" + (numarFirExecutie + 1) +
                " is not-alive (new or dead)", 20, 135 + count * 15);
    }
}
}

```

```

class Ball extends Thread {
    private boolean keepGoing;
    private Graphics g;
    private int x = 37, xChange = 7;
    private int y = 32, yChange = 2;
    private int diameter = 10;
    private int rectLeftX = 30, rectRightX = 130;
    private int rectTopY = 30, rectBottomY = 130;

```

```

public Ball(Graphics graphics) {
    g = graphics;
    keepGoing = true;
}

public void pleaseStop() {
    keepGoing = false;
}

public void run() {
    g.drawRect(rectLeftX, rectTopY, rectRightX - rectLeftX + 10,
        rectBottomY - rectTopY + 10);
    while (keepGoing) {
        g.setColor(Color.white);
        g.fillOval(x, y, diameter, diameter);
        if (x + xChange <= rectLeftX) xChange = - xChange;
        if (x + xChange >= rectRightX) xChange = - xChange;
        if (y + yChange <= rectTopY) yChange = - yChange;
        if (y + yChange >= rectBottomY) yChange = - yChange;
        x = x + xChange;
        y = y + yChange;
        g.setColor(Color.red);
        g.fillOval(x, y, diameter, diameter);
        try {
            Thread.sleep(50);
        }
        catch (InterruptedException e) {
            System.err.println("Sleep exception");
        }
    }
}

```

6.3.4. Prioritatea firelor de execuție

Java definește trei constante pentru selectarea priorităților firelor de execuție:

- public final static int MAX_PRIORITY; //10
- public final static int MIN_PRIORITY; //1
- public final static int NORM_PRIORITY; //5

Mai avem două funcții importante pentru obținerea, respectiv stabilirea priorității:

- public final int getPriority();
- public final void setPriority(int nouaPrioritate);

O altă funcție importantă referitoare la priorități este:

- `public static native void yield();`

Apelul acesteia implică scoaterea procesului curent din execuție și punerea acestuia în coada de așteptare. Funcția `getName()` întoarce numele procesului curent (cărui i s-a dat un nume de tip `String`).

Exemplul 6.3.5. Acest exemplu evidențiază modalitatea de lucru cu prioritățile firelor de execuție:

```
public class Prioritati {
    public static void main(String args[]) {
        // crearea a trei fire de execuție
        FirExecutie fir1 = new FirExecutie("FIR1");
        FirExecutie fir2 = new FirExecutie("FIR2");
        FirExecutie fir3 = new FirExecutie("FIR3");
        // stabilirea priorităților pentru firele de execuție
        fir1.setPriority(Thread.MIN_PRIORITY);
        fir2.setPriority(Thread.MAX_PRIORITY);
        fir3.setPriority(9);
        System.out.println("JVM va crea contextul celor trei" +
            "fire de execuție!");
        // pornirea firelor de execuție
        fir1.start();
        fir2.start();
        fir3.start();
        System.out.println("JVM a creat contextul celor trei" +
            "fire de execuție!");
    }
}

// clasa definită de programator pentru fire de execuție
class FirExecutie extends Thread {
    // constructorul clasei
    public FirExecutie(String nume) {
        super(nume); // apelul metodei superclasei
    }

    public void run() {
        int numarPasi = 5;
        // preia numele firului de execuție
        String numeFir = getName();
        System.out.println("Run se executa de " + numarPasi +
            "ori.");
        for (int i = 1; i <= numarPasi; i++) {
            System.out.println(numeFir + " este la pasul " + i);
        }
    }
}
```

```
try {
    // oprirea pentru 0,5 secunde a firului de execuție
    sleep(500);
} catch (InterruptedException e) {
    System.err.println("Eroare in somn");
}
}
System.out.println(numeFir + " si-a terminat treaba");
}
```

Execuția programului anterior va conduce la afișarea următorului rezultat:

```
JVM va crea contextul celor trei fire de execuție!
Run se executa de 5 ori.
FIR2 este la pasul 1
Run se executa de 5 ori.
FIR3 este la pasul 1
JVM a creat contextul celor trei fire de execuție!
Run se executa de 5 ori.
FIR1 este la pasul 1
FIR2 este la pasul 2
FIR3 este la pasul 2
FIR1 este la pasul 2
FIR2 este la pasul 3
FIR3 este la pasul 3
FIR1 este la pasul 3
FIR2 este la pasul 4
FIR3 este la pasul 4
FIR1 este la pasul 4
FIR2 este la pasul 5
FIR3 este la pasul 5
FIR1 este la pasul 5
FIR2 si-a terminat treaba
FIR3 si-a terminat treaba
FIR1 si-a terminat treaba
```

Remarcăm faptul că la orice pas ordinea de execuție a firelor de execuție este dată de prioritatea aleasă. Se observă că FIR2 este cel mai prioritar, iar FIR1 este cel mai puțin prioritar.

Exemplul 6.3.6. Priorități bazate pe funcția `yield()`, primul fir de execuție lăsând câteodată procesorul pentru celelalte fire de execuție:

```
public class Renuntare {
    public static void main(String args[]) {
```

```

// crearea firelor de executie
FirExecutie fir1 = new FirExecutie("FIR1");
FirExecutie fir2 = new FirExecutie("FIR2");
FirExecutie fir3 = new FirExecutie("FIR3");
// stabilim pentru toate aceeasi prioritate
fir1.setPriority(Thread.NORM_PRIORITY);
fir2.setPriority(Thread.NORM_PRIORITY);
fir3.setPriority(Thread.NORM_PRIORITY);
System.out.println("JVM va crea contextul celor trei" +
    "fire de executie!");
// pornim firele de executie
fir1.start();
fir2.start();
fir3.start();
System.out.println("JVM a creat contextul celor trei" +
    "fire de executie!");
}
}

class FirExecutie extends Thread {
    public FirExecutie(String nume) {
        super(nume);
    }

    public void run() {
        int numarPasi = 5;
        String numeFir = getName();
        System.out.println("Run se executa de " + numarPasi +
            "ori.");
        for (int i = 1; i <= numarPasi; i++) {
            System.out.println(numeFir + " este la pasul " + i);
            // cercetam daca este primul fir
            if (numeFir.equals("FIR1")) {
                // sansa va lua valorile 0 sau 1
                int sansa = (int) (Math.random() * 2);
                System.out.println("sansa = " + sansa);
                if (sansa == 1) yield();
                // uneori cedam procesorul altor fire de executie (2, 3)
            }
            try {
                sleep(500);
            }
            catch (InterruptedException e) {
                System.err.println("Eroare in somn");
            }
        }
    }
}

```

```

}
System.out.println(numeFir + " si-a terminat treaba");
}
}

```

Un rezultat posibil al execuției programului de mai sus este:

```

JVM va crea contextul celor trei fire de executie!
JVM a creat contextul celor trei fire de executie!
Run se executa de 5 ori.
FIR1 este la pasul 1
sansa = 1
Run se executa de 5 ori.
Run se executa de 5 ori.
FIR2 este la pasul 1
FIR3 este la pasul 1
FIR2 este la pasul 2
FIR3 este la pasul 2
FIR1 este la pasul 2
sansa = 1
FIR2 este la pasul 3
FIR3 este la pasul 3
FIR1 este la pasul 3
sansa = 0
FIR2 este la pasul 4
FIR3 este la pasul 4
FIR1 este la pasul 4
sansa = 1
FIR2 este la pasul 5
FIR3 este la pasul 5
FIR1 este la pasul 5
sansa = 1
FIR2 si-a terminat treaba
FIR3 si-a terminat treaba
FIR1 si-a terminat treaba

```

Atunci când sansa ia valoarea 1, primul fir va ceda procesorul în favoarea celorlalte două fire. Acest lucru iese în evidență la primii doi pași din rezultatul obținut anterior.

6.3.5. Metoda sleep

Apelul metodei `sleep()` cere oprirea rulării firului de execuție curent pentru un interval specificat de timp. Există două moduri de apel:

- `public static void sleep(long miliSecunde)`
throws `InterruptedException`;

- `public static void sleep(long miliSecunde, int nanoSecunde)` throws `InterruptedException`;

Ca și `yield()`, metoda `sleep()` este statică, deci acționează direct asupra firului de execuție curent. Aceasta trece firul în starea *Blocked* și va relua execuția lui după sfârșitul perioadei de „adormire” sau dacă se intervine prin apelul metodei `interrupt()`.

6.3.6. Starea de blocare

În general, la metodele care conțin citiri/scrieri sunt posibile blocări.

Exemplul 6.3.7. Fie următorul cod Java:

```
try {
    /** crearea unei conexiuni cu un server */
    Socket sock = new Socket("numeServer", 5000);
    /** obtinerea unui flux de intrare cu ajutorul caruia
       citim datele trimise de server */
    InputStream is = sock.getInputStream();
    /** citirea unui octet */
    int b = is.read();
}
catch (IOException e) { ... }
```

Instrucțiunea `int b = is.read();` încearcă să citească un octet dintr-un flux de intrare conectat la portul 5000 al mașinii numite `numeServer`. Dacă octetul este disponibil, atunci el va fi citit și execuția programului continuă. Dacă nu, atunci apelul lui `read()` trebuie să aștepte. Dacă totuși trece un timp de grație (de exemplu, jumătate de oră) fără ca mașina `numeServer` să furnizeze octetul, atunci firul de execuție curent va fi scos din starea *Running* și dus în starea *Blocked*.

6.3.7. Grupuri de thread-uri

Conceptul de grup de thread-uri este asemănător cu conceptul de director de fișiere. Grupurile de thread-uri sunt utile pentru:

- posibilitatea tratării unitare a mai multor thread-uri care fac parte din grup;
- sprijin în implementarea mecanismului de securitate.

Un grup de thread-uri poate conține și alte grupuri de thread-uri, creând un sistem ierarhic. La rădăcina acestei ierarhii se află un grup de thread-uri numit grup de thread-uri sistem.

Când o aplicație Java este pornită, JVM creează un grup inițial de thread-uri numit grup de thread-uri sistem (grup sistem). În acest grup, JVM creează un alt grup de thread-uri numit grup *main*, în care apoi creează un thread (numit *main*) a cărui funcție `run()` apelează metoda `main()` a aplicației. După aceea, aplicația poate trece

la crearea propriilor sale thread-uri. Implicit, aceste fire de execuție fac parte din grupul de thread-uri *main*.

Java implementează conceptul de grup de thread-uri utilizând clasa `ThreadGroup`. Ca și thread-urile, un `ThreadGroup` are priorități, poate fi de tip *Daemon* și are metode de control al thread-urilor din grup. Există în plus o prioritate maximă. Un fir de execuție care face parte din acest grup nu poate avea nivel de prioritate mai mare decât acest maxim.

Amintim câteva metode importante în clasa `ThreadGroup`:

- `public int activeCount();`
Returnează numărul de thread-uri active din grupul din care face parte respectivul thread, precum și din subgrupurile acestui grup.
- `public int enumerate(Thread list[]);`
- `public int enumerate(ThreadGroup list[]);`
- `public int enumerate(Thread list[], boolean recursiv);`
- `public int enumerate(ThreadGroup list[], boolean recursiv);`

Sunt metode utilizate pentru a obține lista thread-urilor sau grupurilor dintr-un grup de thread-uri și subgrupurile lui. Thread-urile, respectiv grupurile, se vor fi regăsi în tabloul `list` transmis ca parametru al funcției.

Exemplul 6.3.8. Crearea unui grup de fire de execuție:

```
public class GrupThread {
    public static void main(String args[]) {
        ThreadGroup toate = new ThreadGroup("toate firele");
        ThreadGroup jumaGrup = new ThreadGroup(toate, "Juma de fire");
        ThreadGroup jumiGrup = new ThreadGroup(toate, "Jumi de fire");

        FirExecutie fir1 = new FirExecutie(toate, "FIR1");
        FirExecutie fir2 = new FirExecutie(toate, "FIR2");
        FirExecutie fir3 = new FirExecutie(jumaGrup, "FIR3");
        FirExecutie fir4 = new FirExecutie(jumaGrup, "FIR4");
        FirExecutie fir5 = new FirExecutie(jumiGrup, "FIR5");
        FirExecutie fir6 = new FirExecutie(jumiGrup, "FIR6");
        System.out.println(
            "JVM va crea contextul grupului de fire!");
        fir1.start();
        fir2.start();
        fir3.start();
        fir4.start();
        fir5.start();
        fir6.start();
        System.out.println(
            "JVM a creat contextul grupului de fire!");
    }
}
```

```
// dorim sa exemplificam cautarea unui thread intr-un grup
// de thread-uri, sa zicem cu numele "FIR5"
int numarFire = toate.activeCount();
Thread gasit = null;
Thread fire[] = new Thread[numarFire];
toate.enumerate(fire, true); // intreaga ierarhie
int i;
boolean ok = false; // ok = false <=> firul a fost gasit
for (i = 0; i < numarFire; i++)
    if (fire[i].getName().equals("FIR5")) {
        gasit = fire[i];
        ok = true;
        break;
    }
if (ok) {
    System.out.print("Am gasit FIR5 pe pozitia " + i);
    System.out.println(" in cadrul grupului intreg.");
}
else
    System.out.println("Nu am gasit FIR5");
}

class FirExecutie extends Thread {
    public FirExecutie(ThreadGroup grup, String nume) {
        super(grup, nume);
    }

    public void run() {
        int numarPasi = 2;
        System.out.println(getName() + "si-a inceput executia.");
        for (int i = 1; i <= numarPasi; i++) {
            System.out.println(getName() + " este la pasul " + i);
            try {
                sleep(500);
            }
            catch (InterruptedException e) {
                System.err.println("Eroare in somn");
            }
        }
        System.out.println(getName() + " si-a terminat treaba");
    }
}
```

Din exemplul de mai sus se poate vedea faptul că firele de execuție sunt rulate „simultan”. De exemplu, se poate întâmpla ca între cele două mesaje („Am gasit FIR5 pe poziția 4” și respectiv „in cadrul grupului intreg”), care sunt afișate în urma execuției unor instrucțiuni consecutive din cadrul aceluiași fir, să se interpună alte mesaje date de alte fire de execuție (de exemplu, „FIR5 este la pasul 1” etc.).

6.3.8. Stări monitor

Se numește *monitor* un obiect care asigură că o variabilă partajată poate fi accesată într-un moment dat de cel mult un fir de execuție. Deci, un monitor poate bloca și reînvia fire de execuție.

Metoda `wait()` trece un fir de execuție din starea *Running* în starea *Waiting*, iar metodele `notify()` și `notifyAll()` mută firele de execuție afară din starea *Wait*. Cu toate acestea, aceste metode sunt foarte diferite de `suspend()`, `resume()`, `sleep()` și `yield()`. În primul rând, pentru că sunt implementate în clasa `Object`, și nu în clasa `Thread`. În al doilea rând, pentru că pot fi apelate doar în cod sincronizat.

În general, există două abordări pentru implementarea administratorilor de fire de execuție:

- administrare preemptivă (preventivă);
- administrare pe baza cuantei de timp (sau Round-Robin).

Facilitățile discutate până acum au fost preemptive. În administrarea preemptivă, există doar două moduri de a face un fir de execuție să părăsească starea *Running* fără a apela explicit metode cum ar fi `wait()` sau `suspend()`:

- nu poate fi gata de execuție (de exemplu, apelul unei funcții de I/O care se blochează);
- poate fi mutat de CPU (Central Processing Unit) pentru că un alt fir de execuție mai prioritar este în starea *Ready*.

Folosind cuante de timp, un fir de execuție este permis să execute doar pentru un timp precizat. Este apoi mutat în starea *Ready*, în general la coadă, după celelalte fire de execuție care așteaptă. Utilizarea cuantelor de timp asigură că niciodată nu va fi în starea *Running* un proces prioritar, iar celelalte să stea „să aștepte la coadă” (eng. *starvation*). Din păcate, administrarea pe baze de cuante de timp implică crearea unui sistem nedeterminist: adică, la un moment dat, nu putem ști care fir de execuție se execută și pentru cât timp. Acest lucru va implica și posibilitatea ca un program Java, care folosește fire de execuție, să aibă ieșiri (comportări) diferite la execuții diferite pentru aceleași date de intrare. Implementările Java depind de platformă. De exemplu, mașinile Solaris sunt preemptive, mașinile MacIntosh se bazează pe cuante de timp. Platformele Windows au fost preemptive inițial, dar începând cu Java 1.0.2. acestea folosesc cuante de timp.

6.3.9. Excludere mutuală și sincronizarea

Presupunem că două thread-uri incrementează valoarea unui întreg partajat (eng. *shared*). Pot apărea probleme în sensul că cele două thread-uri incrementează în același timp întregul, deci se pot sări valori ale acestuia. Problema se rezolvă dacă cel mult un thread are acces la acea dată la un moment dat. Această cerință se numește excludere mutuală.

Modul în care Java rezolvă excluderea mutuală constă în specificarea metodelor care partajează variabile ca fiind *synchronized* în antetul lor. Aceste metode trebuie să fie din aceeași clasă. Orice fir de execuție, care încearcă să acceseze într-o metodă *synchronized* a unui obiect atât timp cât metoda este folosită, este blocat până când primul thread părăsește metoda.

Codul complet al rezolvării acestei probleme este realizat de clasele *Contor1*, *Contor2*, *NumarPartajat*, *DoiContori* din exemplul de mai jos. Clasa *DoiContori* este derivată din clasa *Applet*. În clasa *NumarPartajat*, am definit metoda care poate fi accesată de cel mult un fir de execuție:

```
public synchronized void increment()
```

Exemplul 6.3.9. Fișierul *DoiContori.java* este:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class DoiContori extends Applet implements ActionListener {
    private Button start;
    private int contor = 0;

    public void init() {
        start = new Button("Start");
        add(start);
        start.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == start) {
            contor++;
            Graphics g = getGraphics();
            NumarPartajat contor12 = new NumarPartajat(g, contor);
            Contor1 contor1 = new Contor1(contor12);
            Contor2 contor2 = new Contor2(contor12);
            contor1.start();
            contor2.start();
        }
    }
}
```

```
}
}
}
```

Fișierul *Contor1.java* este:

```
public class Contor1 extends Thread {
    private NumarPartajat contor1;

    public Contor1(NumarPartajat contor1) {
        this.contor1 = contor1;
    }

    public void run() {
        for (int i = 1; i <= 10; i++)
            contor1.increment();
    }
}
```

Fișierul *Contor2.java* este:

```
public class Contor2 extends Thread {
    private NumarPartajat contor2;

    public Contor2(NumarPartajat contor2) {
        this.contor2 = contor2;
    }

    public void run() {
        for (int i = 1; i <= 10; i++)
            contor2.increment();
    }
}
```

Fișierul *NumarPartajat.java* este:

```
import java.awt.*;

public class NumarPartajat {
    private int n = 0;
    private Graphics g;
    private int x = 0;
    private int contor;

    public NumarPartajat(Graphics g, int contor) {
        this.g = g;
        this.contor = contor;
    }
}
```

```

public synchronized void increment() {
    n = n + 1;
    g.drawString(n + " ", "x * 20, 30 + 15 * contor);
    x++;
}
}

```

6.3.10. Lucrul cu wait() și notify()

Tot o problemă de excludere mutuală este exemplul producător-consumator. Producătorul furnizează date pe care consumatorul le utilizează mai departe. Ce se întâmplă dacă vitezele de producere și consum diferă? Prin `synchronized` am rezolvat problema accesului simultan la o dată (excluderea mutuală). Dacă producătorul este mai rapid, el va citi o dată înscrisă de primul de mai multe ori.

Rezolvarea se realizează cu metodele `wait()`, `notify()` și `notifyAll()` ale clasei `Object`. Apelul lui `wait()` va trece obiectul apelat în starea *Blocked*. El rămâne blocat până la apelul unei metode `notify()` sau `notifyAll()` pentru același obiect. Condiția necesară pentru a se apela una din aceste metode este ca apelul lor să se facă în interiorul metodelor `synchronized`.

Există metodele `wait()` prin care se poate specifica o durată de timp maximă de așteptare. În acest caz, `thread`-ul rămâne blocat până când timpul expiră sau alt `thread` apelează `notify()`.

Exemplul 6.3.10. Dorim să scriem la ecran două texte, fiecare corespunzător unui fir de execuție. Metoda `askFor()` din clasa `ScreenController` joacă rol de monitor (C.A.R. Hoare, 1974). (Monitor înseamnă un obiect special, care permite numai unui singur `thread` să apeleze o metodă a lui, la un moment dat.)

Fișierul `Initial.java` este:

```

import java.applet.Applet;
import java.awt.*;

public class Initial extends Applet {
    public void init() {
        Graphics g = getGraphics();
        ScreenController screen = new ScreenController();

        Text1 text1 = new Text1(g, screen);
        Text2 text2 = new Text2(g, screen);
        text1.start();
        text2.start();
    }
}

```

Fișierul `ScreenController.java` este:

```

public class ScreenController {
    private boolean inUse = false;

    public synchronized void askFor() {
        while (inUse)
            try {
                wait();
            }
            catch (InterruptedException e) {
                System.err.println("Exceptie!");
            }
        inUse = true;
    }

    public synchronized void relinquish() {
        inUse = false;
        notify();
    }
}

```

Fișierul `Text1.java` este:

```

import java.awt.*;

class Text1 extends Thread {
    private Graphics g;
    private ScreenController screen;

    public Text1(Graphics g, ScreenController screen) {
        this.screen = screen;
        this.g = g;
    }

    public void run() {
        while (true) {
            screen.askFor();
            draw();
            screen.relinquish();
        }
    }

    private void draw() {
        g.drawString("Acesta este primul document", 10, 100);
        g.drawString("pe care dorim sa-l afisam la ecran", 10, 115);
    }
}

```

```

    }
}

```

Fișierul Text2.java este:

```

import java.awt.*;

class Text2 extends Thread {
    private Graphics g;
    private ScreenController screen;

    public Text2(Graphics g, ScreenController screen) {
        this.screen = screen;
        this.g = g;
    }

    public void run() {
        while (true) {
            screen.askFor();
            draw();
            screen.relinquish();
        }
    }

    private void draw() {
        g.drawString("Acesta este al doilea document", 10, 25);
        g.drawString("pe care dorim sa-l afisam apoi la ecran", 10,
            40);
    }
}

```

Exemplul 6.3.11. Tot în categoria producător/consumator avem o aplicație ceas (digital) în care am definit două fire de execuție: *Secunde* și *Minute*. Thread-ul *Secunde* este cel care se află aproape tot timpul în secțiunea critică. După 59 de secunde, se apelează `notify()` și thread-ul *Secunde* părăsește secțiunea critică. Thread-ul *Minute*, care „bate la ușa” secțiunii critice, intră acum și incrementează minutele. Codul complet este:

Fișierul Ceas.java este:

```

import java.awt.*;
import java.applet.Applet;

public class Ceas extends Applet {
    public void init() {
        Graphics g = getGraphics();
        TicTac minuteTic = new TicTac();
    }
}

```

```

Minute minute = new Minute(g, minuteTic);
minute.start();
Secunde secunde = new Secunde(g, minuteTic);
secunde.start();
}
}

```

Fișierul Secunde.java este:

```

import java.awt.*;

public class Secunde extends Thread {
    private Graphics g;
    private int secunde = 0;
    private TicTac minuteTic;

    public Secunde(Graphics g, TicTac minuteTic) {
        this.g = g;
        this.minuteTic = minuteTic;
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                System.err.println("Exceptie !");
            }

            if (secunde == 59) {
                minuteTic.tic();
                secunde = 0;
            }
            else secunde++;
            g.clearRect(10, 0, 100, 20);
            g.drawString(secunde + " secunde", 10, 20);
        }
    }
}

```

Fișierul Minute.java este:

```

import java.awt.*;

public class Minute extends Thread {
    private Graphics g;
}

```

```

private int minute = 0;
private TicTac minuteTic;

public Minute(Graphics g, TicTac minuteTic) {
    this.g = g;
    this.minuteTic = minuteTic;
}

public void run() {
    while (true) {
        minuteTic.waitForTic();
        if (minute == 59)
            minute = 0;
        else
            minute++;
        g.clearRect(10, 20, 100, 20);
        g.drawString(minute + " minute", 10, 40);
    }
}

```

Fișierul TicTac.java este:

```

public class TicTac {
    private boolean ticHappens = false;

    public synchronized void waitForTic() {
        while (!ticHappens) {
            try {
                wait();
            }
            catch (InterruptedException e) {
                System.err.println("Exceptie la TicTac !");
            }
            ticHappens = false;
        }
    }

    public synchronized void tic() {
        ticHappens = true;
        notify();
    }
}

```

Exemplul 6.3.12. Un program Java pentru simularea unei cafenele. Informațiile sunt trimise de la un thread la altul. O comandă se realizează prin apăsarea unui buton

(burger, fries, cola). Comanda se afișează pe ecran și apoi se introduce într-o coadă care se afișează pe ecran. Șeful ia câte o comandă din coadă în sistemul primul venit, primul servit. Chelnerul acceptă o comandă, o afișează și o inserează în coadă.

Fișierul Cafe.java este:

```

import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Cafe extends Applet implements ActionListener {
    private Button burger, fries, cola, cooked;
    private Order order, complete;

    public void init() {
        Graphics g = getGraphics();
        burger = new Button("Burger");
        add(burger);
        burger.addActionListener(this);
        fries = new Button("Fries");
        add(fries);
        fries.addActionListener(this);
        cola = new Button("Cola");
        add(cola);
        cola.addActionListener(this);
        cooked = new Button("Cooked");
        add(cooked);
        cooked.addActionListener(this);
        order = new Order();
        Queue queue = new Queue(g);
        complete = new Order();
        Waiter waiter = new Waiter(g, order, queue);
        waiter.start();
        Chef chef = new Chef(g, complete, queue);
        chef.start();
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == burger)
            order.notifyEvent("burger");
        if (event.getSource() == fries)
            order.notifyEvent("fries");
        if (event.getSource() == cola)
            order.notifyEvent("cola");
        if (event.getSource() == cooked)

```

```
complete.notifyEvent("cooked");
```

```
}  
}
```

Fişierul Waiter.java este:

```
import java.awt.*;  
  
public class Waiter extends Thread {  
    private Order order;  
    private Graphics g;  
    private Queue queue;  
  
    public Waiter(Graphics g, Order order, Queue queue) {  
        this.order = order;  
        this.g = g;  
        this.queue = queue;  
    }  
  
    public void run() {  
        g.drawString("O noua comanda", 10, 50);  
        while (true) {  
            String nouaComanda = order.waitForEvent();  
            g.clearRect(10, 50, 50, 25);  
            g.drawString(nouaComanda, 10, 70);  
            try {  
                Thread.sleep(5000);  
            }  
            catch (InterruptedException e) {  
                System.err.println("Exceptie Waiter !");  
            }  
            if (!queue.isFull())  
                queue.enter(nouaComanda);  
            queue.enter(nouaComanda);  
        }  
    }  
}
```

Fişierul Order.java este:

```
public class Order {  
    private String order = "";  
  
    public synchronized void notifyEvent(String nouaComanda) {  
        order = nouaComanda;  
        notify();  
    }  
}
```

```
}  
  
public synchronized String waitForEvent() {  
    while (order.equals(""))  
        try {  
            wait();  
        }  
        catch (InterruptedException e) {  
            System.err.println("Exceptie Order!");  
        }  
    String nouaComanda = order;  
    order = "";  
    return nouaComanda;  
}  
}
```

Fişierul Queue.java este:

```
import java.awt.*;  
  
public class Queue {  
    private Graphics g;  
    private String[] queue = new String[20];  
    private int count = 0;  
  
    public Queue(Graphics g) {  
        this.g = g;  
    }  
  
    public synchronized void enter(String item) {  
        queue[count] = item;  
        count++;  
        display();  
        notify();  
    }  
  
    public synchronized String remove() {  
        while (count == 0)  
            try {  
                wait();  
            }  
            catch (InterruptedException e) {  
                System.err.println("Exceptie Queue!");  
            }  
        String item = queue[0];  
        queue[0] = null;  
        count--;  
        display();  
        notify();  
        return item;  
    }  
}
```

```

count --;
for (int c = 0; c < count; c++)
    queue[c] = queue[c + 1];
display();
return item;
}

public synchronized boolean isFull() {
    return count == queue.length;
}

private void display() {
    g.drawString("Coada", 120, 50);
    g.clearRect(120, 50, 50, 220);
    for (int c = 0; c < count; c++)
        g.drawString(queue[c], 120, 70 + c * 20);
}
}

```

Fișierul Chef.java este:

```

import java.awt.*;

public class Chef extends Thread {
    private Graphics g;
    private Order complete;
    private Queue queue;

    public Chef(Graphics g, Order complete, Queue queue) {
        this.g = g;
        this.complete = complete;
        this.queue = queue;
    }

    public void run() {
        g.drawString("Cooocking", 200, 50);
        while (true) {
            String order = queue.remove();
            g.clearRect(200, 55, 50, 25);
            g.drawString(order, 200, 70);
            String cookedInfo = complete.waitForEvent();
            g.clearRect(200, 55, 50, 25);
        }
    }
}

```

6.4. Studii de caz: problema filosofilor și problema producător-consumator

6.4.1. Problema filosofilor

Cinci filosofi stau la o masă rotundă unde se află corespunzător cinci farfurii și doar cinci furculițe (de fapt, bețe chinezești). Filosofii pot face două lucruri: să mănânce sau să mediteze. Un filosof poate mânca din farfuria din fața sa dacă acesta poate lua cele două bețișoare din stânga și din dreapta. Descrieți un algoritm care să rezolve problema astfel încât:

- doi filosofi nu pot deține aceeași furculiță simultan;
- nu apare deadlock;
- nici un filosof nu moare de foame;
- dacă există filosofi care suferă (temporar) de anorexie, ceilalți să nu fie afectați.

Folosind monitoare, iată o soluție posibilă în pseudocod:

```

// definitia monitorului
monitor MonitorFurculite {
    // furculita[i] = numarul de furculite de care dispune
    // filosoful i.
    furculita : array (0 .. 4) of Integer range 0 .. 2 = (2, 2,
        2, 2, 2)
    // Condition este variabila de conditie
    OK_mananca : array (0 .. 4) of Condition;

    procedure IaFurculite(i : Integer) {
        if furculita(i) <> 2 then Wait(OK_mananca(i));
        furculita((i + 1) mod 5) := furculita((i + 1) mod 5) - 1;
        furculita((i - 1) mod 5) := furculita((i - 1) mod 5) - 1;
    }

    procedure LasaFurculite(i : Integer) {
        furculita((i + 1) mod 5) := furculita((i + 1) mod 5) + 1;
        furculita((i - 1) mod 5) := furculita((i - 1) mod 5) + 1;
        if furculita((i+1) mod 5) = 2 then Signal(OK_mananca
            ((i+1) mod 5));
        if furculita((i-1) mod 5) = 2 then Signal(OK_mananca
            ((i-1) mod 5));
    }
} // sfarsit definitie MonitorFurculite

// definitia proceselor filosof
proces Filosof(i) {

```



```

while (true) {
    gandeste;
    IaFurculite(i);
    mananca;
    LasaFurculite(i);
}
}

```

Codul sursă Java este dat în continuare. Fișierul sursă CinciFilosofi.java este:

```

import java.applet.Applet;
import java.awt.*;

// definitia clasei CinciFilosofi extinsa din clasa Applet.
public class CinciFilosofi extends Applet {
    public void init() {
        // setam un font vizibil.
        Font f = new Font("Courier", Font.BOLD, 50);
        setFont(f);
        // setam dimensiunea appletului
        setSize(400,350);
        // preluam in variabila g contextul grafic curent.
        Graphics g = getGraphics();
        // declaram si definim obiectul masa ce reprezinta
        // sectiunea critica.
        Furculite masa = new Furculite();
        // declaram si definim un vector de fire de executie
        // corespunzator celor 5 filosofi.
        Thread filosof[] = new Thread[5];
        int i; // variabila de lucru.
        for (i = 0; i < filosof.length; i++) {
            // primul argument al constructorului este numele
            // firului de executie curent;
            // al doilea argument este contextul grafic;
            // al treilea este masa (sectiunea critica);
            // al patrulea si al cincilea argument reprezinta
            // cele doua furculite. Pentru al cincilea filosof
            // trebuie facut modulo 5 (pentru furculita a doua).
            filosof[i] = new Filosof("Filosoful " + (i + 1), g,
                                    masa, i, (i + 1) % filosof.length);
        }
        // startam cele 5 fire de executie care vor lucra
        // in paralel.
        for (i = 0; i < filosof.length; i++)
            filosof[i].start();
    } // sfarsit public void init().
}

```

```

} // sfarsit public class CinciFilosofi extends Applet.

```

```

class Filosof extends Thread {
    // g reprezinta contextul grafic.
    private Graphics g;
    // masa reprezinta sectiunea critica.
    private Furculite masa;
    // fUnu si fDoi reprezinta cele doua furculite.
    private int fUnu, fDoi;

    // constructorul clasei Filosof.
    public Filosof(String nume, Graphics g, Furculite masa,
                  int fUnu, int fDoi) {
        // se apeleaza constructorul Thread(nume) din clasa
        // de baza Thread. Numele thread-ului se poate acum
        // obtine cu getName().
        super(nume);
        // folosind referinta this se definesc valorile datelor
        // private din clasa Filosof.
        this.g = g;
        this.masa = masa;
        this.fUnu = fUnu;
        this.fDoi = fDoi;
    } // sfarsitul definitiei constructorului clasei Filosof.

    // definitia metodei run() ce trebuie suprascrisa. run()
    // este metoda abstracta in interfata Runnable. Clasa
    // Thread o implementeaza.
    public void run() {
        // definim numeFilosof prin apelul metodei getName()
        // din clasa Thread.
        String numeFilosof = getName();
        // obtinem ultimul caracter din numele Thread-ului.
        // Acesta este de fapt numarul filosofului.
        char numarCharFilosof = numeFilosof.charAt(10);
        // convertim un char la un int.
        int numarFilosof = (int) numarCharFilosof - '0';
        // presupunem ca filosofii mananca sau gandesc pentru
        // totdeauna.
        while (true) {
            // facem apel la metoda cereFurculitele din clasa
            Furculite.
            masa.cereFurculitele(fUnu, fDoi);
            // stergem bucata de ecran pe care vom afisa apoi masa

```

```

// ca filosoful curent mananca!
g.clearRect(15, 60 * (numarFilosof - 1) + 10, 800, 70);
g.drawString(umeFilosof + " mananca!", 10, 60 *
    numarFilosof);
// i-1 lasam 100 milisecunde sa manance.
try {
    Thread.sleep(100);
}
catch (InterruptedException e) {
    System.err.println("Exceptie de tip Sleep: " +
        umeFilosof);
}
// dupa ce "s-a saturat" de mancat, apelam metoda
// elibereazaFurculitele() din clasa Furculite.
masa.elibereazaFurculitele(fUnu, fDoi);
// stergem bucata de ecran pe care vom afisa apoi
// mesajul ca filosoful curent se gandeste!
g.clearRect(15, 60 * (numarFilosof - 1) + 10, 800, 70);
g.drawString(umeFilosof + " gandeste!", 10, 60 *
    numarFilosof);
}
} // sfarsitul definitiei metodei run().
} // sfarsitul definitiei clasei Filosof.

```

Fișierul Furculite.java este:

```

// definitia clasei Furculite ce implementeaza sectiunea
// critica a problemei filosofilor.
public class Furculite {
    // declaram un vector de 5 furculite.
    private boolean[] furculita = new boolean[5];

    // constructorul clasei.
    public Furculite() {
        // la inceput, toate furculitele sunt "pe masa".
        // De aceea, le initializam cu false. Astfel,
        // furculita[i] = true <=> furculita i este
        // ridicata de un filosof.
        for (int i = 0; i < 5; i++)
            furculita[i] = false;
    }

    // metoda cereFurculitele de parametri fUnu si fDoi.
    // Aceasta este etichetata synchronized astfel incat
    // cel mult un Thread o executa la un moment dat.

```

```

public synchronized void cereFurculitele(int fUnu, int fDoi) {
    // atata timp cat furculita "din stanga" sau cea
    // "din dreapta" sunt ridicate de alt filosof,
    // asteapta la coada.
    while ((furculita[fUnu]) || (furculita[fDoi]))
        try {
            // asteptarea se face prin apelul lui wait() din clasa
            // Object care va trece obiectul apelat in starea Blocked.
            wait();
        }
        catch (InterruptedException e) {
            System.err.println("Exceptie de la cerut furculitele !");
        }
    // dupa ce cele doua furculite au fost puse pe masa, atunci
    // filosoful curent le ridica.
    furculita[fUnu] = true;
    furculita[fDoi] = true;
} // sfarsitul definitiei metodei cereFurculitele().

// metoda elibereazaFurculitele de parametri fUnu si fDoi.
// Aceasta este etichetata synchronized astfel incat
// cel mult un Thread o executa la un moment dat.
public synchronized void elibereazaFurculitele(int fUnu, int
    fDoi) {
    // filosoful curent a terminat de mancat si lasa pe masa
    // cele doua furculite.
    furculita[fUnu] = false;
    furculita[fDoi] = false;
    // apelam metoda notify() din clasa Object pentru a scoate
    // primul obiect din coada de asteptare Ready.
    notify();
} // sfarsitul definitiei metodei elibereazaFurculitele().
} // sfarsitul definitiei clasei Furculite.

```

6.4.2. Problema producător-consumator

În anumite tipuri de aplicații se poate întâmpla ca o porțiune de program să producă valori care apoi sunt consumate de o porțiune diferită a aceluiași program. Un astfel de aranjament se numește relație producător-consumator. Java furnizează un mod elegant de organizare a acestui tip de program prin folosirea firelor de execuție multiple și pipe-uri. Producătorul și consumatorul rulează fiecare propriul fir de execuție, comunicând printr-un pipe.

Un pipe este o zonă de date de tip buffer folosită atât pentru citiri, cât și pentru scrieri. Un pipe poate memora doar un număr limitat de valori. Atât citirea, cât și

scrierea într-un pipe implică suspendarea temporară a unui thread. O scriere va fi suspendată, dacă bufferul pipe curent este plin, în timp ce o operație de citire va fi suspendată, dacă bufferul este vid. În ambele cazuri, execuția va continua dacă condiția este rezolvată (când cititorul eliberează spațiu din buffer, respectiv când scriitorul adaugă un element în buffer).

În Java, pipe-urile sunt reprezentate în clasele `PipedInputStream`, respectiv `PipedOutputStream`. Fiecare pipe se manifestă printr-o pereche de pointeri la fluxuri. Valoarea a doua creată (pipe-ul de intrare sau ieșire) are ca argument prima valoare și astfel se creează o conexiune între două pipe-uri:

```
PipedInputStream intrare = new PipedInputStream();
PipedOutputStream iesire = new PipedOutputStream(intrare);
```

Valorile pot fi scrise secvențial într-un flux pipe de ieșire, ca și în alte tipuri de fluxuri de ieșire. Aceste valori pot fi citite din fluxul pipe de intrare în aceeași ordine cum au fost inserate.

Problema pe care o discutăm este cea a generării numerelor mai mici decât un număr precizat (de exemplu, 1.000.000) care sunt prime și numere Fibonacci (în același timp). Amintim că:

- un număr este prim, dacă nu are alți divizori decât 1 și el însuși;
- un număr Fibonacci este dat de recurența liniară de ordin 2:
 $f_0 = 0; f_1 = 1; \text{ (uneori se alege } f_0 = 1 \text{)}$
 $f_n = f_{n-2} + f_{n-1};$

Pentru fiecare dintre aceste secvențe de numere este creat câte un fir de execuție. Un thread, de fapt, nu are nevoie să știe că aceasta are de-a face cu un pipe. Firul *GenereazaFibonacci* creează o secvență de valori din șirul lui Fibonacci pe care apoi le scrie într-un flux de ieșire. Folosind un `DataOutputStream`, putem scrie ușor valori întregi. Firul *GenereazaPrim* creează o secvență de valori numere prime între 2 și un număr maxim precizat aprioric. Programul principal (*PipeTest*) va crea și va porni cele două fire de execuție în metodele:

```
private DataInputStream apelFibonacci(int maximNumere)
```

și:

```
private DataInputStream apelPrim(int maximNumere)
```

Acestea vor face și legătura dintre fluxurile de intrare/ieșire în pipe-ul respectiv (cum am discutat mai sus):

Constructorul clasei *PipeTest* va citi din pipe-uri câte un întreg natural reprezentând numărul Fibonacci curent sau numărul prim curent. Dacă sunt egale, atunci se va afișa un mesaj precum că numărul citit este Fibonacci și prim în același timp. Dacă numărul Fibonacci citit din pipe-ul Fibonacci este mai mic decât numărul prim citit din pipe-ul numerelor prime, atunci se va citi doar din pipe-ul Fibonacci (în caz contrar, din pipe-ul numerelor prime). Execuția programului se încheie când numărul Fibonacci curent este mai mic decât un număr maxim curent. Se preferă comparația

cu Fibonacci (și nu cu cele prime), deoarece acestea sunt mai rare (adică diferența dintre două numere Fibonacci este mai mare decât diferența dintre două numere prime consecutive).

Observație: Două rezultate din teoria numerelor:

Se cunoaște teorema de distribuție a numerelor prime. Notând cu p_n al n-lea număr prim, se știe că limita $p_n/n \cdot \ln n$ este 1, pentru n tinzând la infinit. Cât despre șirul lui Fibonacci, lucrurile sunt mai clare. Se cunoaște chiar formula exactă a termenului general. Notând cu f_n al n-lea număr Fibonacci, avem $f_n = (1/\sqrt{5}) * [((1+\sqrt{5})/2)^{n+1} - ((1-\sqrt{5})/2)^{n+1}]$. Atât p_n , cât și f_n , sunt strict crescătoare. În plus, se observă că șirul f_n este mai rapid crescător decât p_n (deoarece $\lim p_n/f_n = 0$).

Codul sursă Java este:

```
import java.io.*;

// definim clasa GenereazaFibonacci derivata din Thread
// aceasta are drept scop afisarea si transmiterea catre Pipe
// a urmatorului numar Fibonacci
class GenereazaFibonacci extends Thread
{
    private DataOutputStream iesire;
    private int maximNumere;

    public GenereazaFibonacci(DataOutputStream iesire, int
maximNumere) {
        this.iesire = iesire;
        this.maximNumere = maximNumere;
    }

    public void run() {
        // initializam primele doua valori din sirul
        // lui Fibonacci
        int n = 0;
        int m = 1;
        try {
            iesire.writeInt(m);
            int nouaValoare;
            while (m < maximNumere) {
                nouaValoare = n + m;
                n = m;
                m = nouaValoare;
                System.out.println("Noul numar Fibonacci = " +
nouaValoare);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        iesire.writeInt(nouaValoare);
    }
    iesire.close();
}
catch (IOException e) {
    return;
}
} // de la metoda public void run()
} // sfarsitul clasei GenereazaFibonacci

// definim clasa GenereazaPrim derivata din Thread
// aceasta are drept scop afisarea si transmiterea catre Pipe
// a urmatorului numar prim
class GenereazaPrim extends Thread {
    private DataOutputStream iesire;
    private int maximNumere;

    public GenereazaPrim(DataOutputStream iesire, int
maximNumere) {
        this.iesire = iesire;
        this.maximNumere = maximNumere;
    }

    public void run() {
        // initializam nouaValoare cu 1
        int nouaValoare = 1;
        boolean estePrim; // estePrim = true <=> numarul curent
                           // este prim
        int i; // variabila de lucru
        try {
            while (nouaValoare < maximNumere) {
                nouaValoare++;
                estePrim = true;
                for (i = 2; i * i <= nouaValoare; i++)
                    if (nouaValoare % i == 0) {
                        estePrim = false;
                        break;
                    }
                if (estePrim) {
                    System.out.println("Noul numar prim = " + nouaValoare);
                    iesire.writeInt(nouaValoare);
                }
            }
            iesire.close();
        }
    }
}

```

```

        catch (IOException e) {
            return;
        }
    } // de la metoda public void run()
} // sfarsitul clasei GenereazaPrim

// definim clasa PipeTest care va folosi un pipe pentru
// comunicarea dintre cele doua fire de executie
public class PipeTest {
    static public void main(String [] args) {
        BufferedReader in;
        String linie;
        int maximNumere = 0;
        try {
            in = new BufferedReader(new InputStreamReader(
                System.in), 1);
            System.out.flush();
            System.out.println("Dati numarul maxim de numere: ");
            linie = in.readLine();
            maximNumere = Integer.parseInt(linie);
            in.close();
        }
        catch (IOException e) {
            System.out.println("Citire gresita de la tastatura " +
                e.toString());
        }
        PipeTest obiect = new PipeTest(System.out, maximNumere);
    } // sfarsitul metodei static public void main(String [] args)

    private PipeTest(PrintStream iesire, int maximNumere) {
        DataInputStream fibonacci = apelFibonacci(maximNumere);
        DataInputStream prim = apelPrim(maximNumere);
        try {
            int x = fibonacci.readInt();
            int y = prim.readInt();
            while (x < maximNumere) {
                if (x == y) {
                    iesire.println("numarul " + x +
                        " este fibonacci si prim !!");
                    x = fibonacci.readInt();
                    y = prim.readInt();
                }
                else

```

```

        if (x < y)
            x = fibonacci.readInt();
        else
            y = prim.readInt();
    }
}
catch (IOException e) {
    System.exit(0);
}
} // sfarsitul definitiei metodei private PipeTest()

private DataInputStream apelFibonacci(int maximNumere) {
    try {
        // cream un generator de numere Fibonacci
        PipedInputStream intrare = new PipedInputStream();
        PipedOutputStream iesire = new PipedOutputStream(
            intrare);
        Thread firFibonacci = new GenereazaFibonacci(
            new DataOutputStream(iesire), maximNumere);
        firFibonacci.start();
        return new DataInputStream(intrare);
    }
    catch (IOException e) {
        return null;
    }
} // sfarsitul metodei private apelFibonacci()

private DataInputStream apelPrim(int maximNumere) {
    try {
        // cream un generator de numere prime
        PipedInputStream intrare = new PipedInputStream();
        PipedOutputStream iesire = new PipedOutputStream(
            intrare);
        Thread firPrim = new GenereazaPrim(
            new DataOutputStream(iesire), maximNumere);
        firPrim.start();
        return new DataInputStream(intrare);
    }
    catch (IOException e) {
        return null;
    }
} // sfarsit metoda private DataInputStream apelPrim()
} // sfarsitul clasei public class PipeTest

```

6.5. Concluzii

Firele de execuție permit executarea simultană a mai multor secvențe de cod. Firele de execuție pot partaja aceleași resurse, iar accesul la acestea poate fi concurrent sau secvențial. Pentru a nu apărea probleme de consistență, este bine ca modificarea datelor comune să se realizeze secvențial. Acest lucru este posibil datorită sistemului de sincronizare pus la dispoziție de mediul Java.

Sistemul pe care sunt executate programele multithread nu trebuie să fie neapărat multiprocesor. Sistemele de operare existente se ocupă de gestiunea proceselor și a firelor de execuție.

În funcție de importanța operațiilor efectuate, un fir de execuție poate fi mai prioritar decât altele la primirea resurselor necesare rulării de la sistemul de operare.

Mai multe fire de execuție pot fi grupate pentru a fi tratate unitar. Un grup poate conține la rândul său și alte grupuri de fire de execuție.

6.6. Test grilă

Întrebarea 6.6.1. Care dintre afirmațiile următoare sunt false?

- Un fir de execuție nu poate crea un alt fir de execuție.
- O aplicație poate crea cel mult 10 fire de execuție.
- Două fire de execuție nu pot citi simultan date din același fișier.
- Firele de execuție nu pot avea obiecte comune.
- Pentru fiecare fir de execuție trebuie să definim o clasă.

Întrebarea 6.6.2. Cuvântul cheie `synchronized` se referă la:

- sincronizarea comunicării între firele de execuție;
- specificarea datelor care vor fi accesibile tuturor firelor de execuție;
- excluderea mutuală;
- proprietatea de continuitate.

Întrebarea 6.6.3. Sincronizarea firelor de execuție se realizează cu ajutorul:

- cuvântului cheie `synchronized`;
- datelor membre statice;
- metodelor `wait()`, `notify()` și `notifyAll()`;
- metodelor `sleep()`, `notify()` și `notifyAll()`.

Întrebarea 6.6.4. Un fir de execuție poate intra în starea de blocare astfel:

- prin apelul metodei `sleep()`;
- automat de către sistemul de operare;
- prin apelul metodei `block()`;
- prin apelul metodei `wait()`.

6.7. Exerciții propuse spre implementare

Exercițiul 6.7.1. Scrieți un program Java (care folosește fire de execuție) pentru simularea unui cronometru. Acesta are două butoane, unul pentru start și continuarea secvenței de timp și un al doilea pentru revenirea acului indicator la poziția inițială.

Exercițiul 6.7.2. Scrieți un program Java care generează două fire de execuție pentru parcurgerea unui `String` de la cele două capete. Folosiți doi pointeri (`ptrStg` și `ptrDr`) a căror valoare se incrementează, respectiv decrementează într-o funcție din memoria comună (declarată într-o funcție `synchronized`). În memoria comună se află `String`-ul. Firele de execuție se întâlnesc la „mijloc”.

Exercițiul 6.7.3. Scrieți un applet Java care generează la apăsarea unui buton o minge colorată (distinct de precedentele) care se „izbește” și „ricșează” de marginile unui dreptunghi. Se vor genera cel mult N mingi (N , de exemplu, 20). Apăsând un alt buton, acestea se pot opri din mișcarea lor (în ordinea inversă generării lor). Eventual se va permite ca mingile să se ciocnească.

Exercițiul 6.7.4. Scrieți un applet Java care poate genera trei mingi colorate cu roșu, galben și albastru atunci când se apasă câte un buton (eventual denumite *roșu*, *galben* și *albastru*). Aceste mingi se „izbesc” și „ricșează” într-un dreptunghi dat. Dacă se reapasă un buton precedent, atunci mingea generată anterior se oprește, apărând în locul ei alta de aceeași culoare. Furnizați câte un buton *Stop* pentru fiecare dintre aceste 3 culori menit să oprească mișcarea mingii de culoarea respectivă. Puneți la dispoziție și câte o bară de defilare necesară pentru modificarea vitezei mingilor colorate.

6.8. Proiecte propuse spre implementare

Proiectul 6.8.1. (Problema filosofilor) Rescrieți programul de la problema filosofilor (vezi secțiunea 5.4.1.) în cazul în care bețișoarele sunt în centrul mesei.

Proiectul 6.8.2. (Breakout) Scrieți un program Java, care folosește fire de execuție, pentru simularea jocului Breakout. Acesta constă dintr-un suport care se mișcă orizontal având la început o bilă situată pe el. Deasupra suportului (la o anumită distanță mai sus) există un zid din cărămizi care urmează a fi doborât de către bilă. În mod obișnuit, bila poate sparge doar o cărămidă. Conform cu poziția și unghiul unde bila lovește suportul, aceasta va ricoșa înapoi către zid sub un unghi egal cu unghiul de cădere, dar în sens suplimentar (invers). Dacă nu se reușește „prinderea” bilei de către suport, atunci acea bilă își încetează mișcarea pe ecran. Cărămizile pot fi obișnuite, pot conține bonusuri (de exemplu, dacă reușiți să prindeți bila care a atins cărămidă bonificată, atunci bila va sparge întreaga lățime a zidului) sau pot genera mai multe bile.

7. Appleturi și instrumente de lucru cu ferestrele (AWT)

7.1. Cuvinte cheie

- appleturi, navigator, pagină Web
- AWT (*Abstract Window Toolkit*)
- componente grafice și evenimente
- gestionari de poziționare, ferestre
- manager de securitate

7.2. Appleturi

Am văzut în capitolul 1 că appleturile sunt încărcate de pe un calculator aflat la distanță (server) și apoi executate local (în navigatorul Web). Din punctul de vedere al securității, appleturile sunt foarte restrictive relativ la următoarele operații:

- nu pot rula nici un program executabil local;
- nu pot citi sau scrie în sistemul de fișiere al calculatorului local;
- pot comunica doar cu serverul original;
- pot afla doar o mulțime restrictivă de lucruri despre calculatorul local. De exemplu, pot determina numele și versiunea sistemului de operare, dar nu și numele utilizatorului sau adresa acestuia de e-mail.

7.2.1. Clasa Applet (java.applet.Applet)

Un applet este un mic program Java care este menit pentru inserare în alte aplicații. Clasa Applet trebuie să fie superclasa oricărui applet care va fi înglobat în paginile Web (sau rulat de appletviewer). Această clasă face parte din pachetul java.applet și extinde clasa Panel (care la rândul ei are ca superclase Container, Component și Object).

Un applet este introdus în cadrul unei pagini Web prin intermediul marcatorului <applet>. Acesta poate avea următoarele atribute:

Nume atribut	Valoare atribut
archive	Enumerare de arhive separate prin virgulă.
code	Conține numele clasei appletului.
codebase	URL-ul directorului în care se află fișierul .class corespunzător appletului.
object	Indică un fișier care conține un applet serializat.
alt	Specifică un text care va fi afișat de navigatoarele care nu suportă appleturi.
name	Stabilește un nume pentru applet.
align	Indică modalitatea de aliniere (este similar cu atributul align de la marcatorul și poate avea valorile: top, center, bottom).
width	Stabilește lățimea appletului în pixeli.
height	Stabilește înălțimea appletului în pixeli.
vspace	Indică distanța pe verticală care este lăsată înainte și după applet.
hspace	Indică marginea care este lăsată în stânga și dreapta appletului.

Exemplul 7.2.1. Un applet poate fi inserat într-o pagină astfel:

```
<applet code="TestApplet.class"
```

```
width="300" height="400"
vspace="30" hspace="50"
alt="Un applet pentru test." align="top">
</applet>
```

Astfel, appletul va avea 300 pixeli lungime și 400 înălțime, se va lăsa o margine de 30 pixeli înainte și 30 după, 50 la stânga și 50 la dreapta. Dacă marcatorul va fi succedat de un text, acesta va apărea lângă applet în partea de sus. Dacă navigatorul nu suportă appleturi, atunci se va afișa textul stabilit de tagul alt.

Cele mai importante metode din clasa Applet sunt următoarele:

Prototipul metodei	Descrierea metodei
public void init()	Este prima metodă apelată de navigator sau de applet viewer. Apoi se apelează metoda start(). Există navigatoare care apelează această metodă de mai multe ori.
public void start()	Se apelează imediat după metoda init() și la reluarea execuției appletului (de exemplu, când se revine în pagină).
public void stop()	Se apelează la suspendarea activității appletului. Unele navigatoare nu apelează această metodă.
public void destroy()	Se apelează la terminarea existenței appletului.

Implicit, apelul acestor metode nu are efect. Spre a asocia anumite acțiuni pentru applet, va trebui să redefinim metodele prezentate în tabelul anterior. De asemenea, pentru a putea desena pe suprafața appletului, va trebui să redefinim și metoda paint(), care este moștenită din clasa Container. Metoda are un singur parametru de tip java.awt.Graphics, care se mai numește și *contextul grafic*. Ori de câte ori se dorește desenarea appletului, se va apela metoda paint(). Aceasta este apelată de navigator sau de appletviewer. Mai multe informații privitoare la desenare se găsesc în subcapitolul 7.3.

Exemplul 7.2.2. Următorul applet evidențiază momentele când sunt apelate metodele discutate anterior. Este bine să încercați acest exemplu cu ajutorul unui navigator și să încercați operațiile de minimizare, maximizare.

```
import java.applet.Applet;
import java.awt.*;

public class TestApplet extends Applet {
    String buf="";

    public void init() {
        buf+="init(); ";
        System.out.println("Appletul s-a initializat!");
    }

    public void paint(Graphics g) {
```

```

    g.drawString(buf, 10, 15);
    buf+=" - ";
    System.out.println("Appletul s-a desenat!");
}
public void start() {
    buf+="start(); ";
    System.out.println("Appletul s-a pornit!");
}
public void stop() {
    buf+="stop(); ";
    System.out.println("Appletul s-a oprit!");
}
public void destroy() {
    System.out.println("Appletul s-a terminat!");
}
}

```

Afișarea în suprafața de desenare a appletului se realizează doar atunci când se apelează metoda `paint()`. Apelurile `println()` afișează mesajele la consolă (pentru a vedea consola trebuie să umblăm la opțiunile de setare a navigatorului, iar dacă avem mediul JDK 1.4 instalat, va trebui să apară în *System tray bar*).

Clasa `Applet` conține câteva metode utile, cum ar fi:

Prototipul metodei	Descrierea metodei
<code>public String getAppletInfo()</code>	Returnează informații despre applet. Implicit returnează <code>null</code> , însă programatorul poate redefini metoda pentru a furniza numele autorului, termenii de copyright și versiunea appletului.
<code>public URL getCodeBase()</code>	Returnează URL-ul directorului în care se găsește appletul.
<code>public URL getDocumentBase()</code>	Returnează URL-ul documentului HTML în care este înglobat appletul.
<code>public boolean isActive()</code>	Indică dacă appletul este activ. Apelul metodei <code>start()</code> face ca appletul să fie activ, iar metoda <code>stop()</code> inactivează appletul.
<code>public void resize(int lungime, int inaltime)</code>	Redimensionează suprafața de afiliere a appletului.
<code>public void showStatus(String mesaj)</code>	Afișează mesajul specificat în bara de stare a navigatorului Web.

7.2.2. Obținerea parametrilor de intrare a appleturilor

Java pune la dispoziție citirea parametrilor pentru un applet din fișierul HTML asociat. În acest fișier, în interiorul tagului `<applet>` se pot scrie linii de forma:

```
<param name="numeSir" value="valoareaSir">
```

Exemplul 7.2.3. Specificarea parametrilor unui applet în fișierul HTML se poate realiza astfel:

```

<applet class="Something.class">
  <param name="arg1" value="test">
  <param name="arg2" value="100">
</applet>

```

Acești parametri se pot obține în applet prin apelarea metodei `getParameter()` care are un argument de tip `String`, care trebuie să fie numele unui parametru (cel care este specificat de atributul `name`) și întoarce valoarea acestuia (dată de atributul `value` corespunzător) care este tot de tip `String`.

Valoarea parametrului `arg1` din exemplul 7.2.1. poate fi obținută astfel:

```
String valoareArg1 = getParameter("arg1");
```

Se va obține șirul "test".

Dacă în fișierul HTML nu găsim nici un parametru cu numele respectiv, atunci `getParameter()` întoarce valoarea `null`.

Exemplul 7.2.4. Testarea, dacă un parametru a fost specificat, se poate realiza astfel:

```

s = getParameter("arg1");
if (s == null) s = "Valoare implicita";

```

sau

```

if ((s = getParameter("arg1")) == null)
    s = "Valoare implicita";

```

Exemplul 7.2.5. Prezentăm un applet care afișează valoarea a trei parametri: `arg1`, `arg2` și `arg3`. Fișierul HTML are următorul conținut:

```

<html>
<head>
  <title>Un exemplu de obtinere a parametrilor</title>
</head>

<body>
  <applet code="GetParam.class" width="400" height="100">
    <param name="arg1" value="Primul parametru">

```



```

    <param name="arg2" value="2">
  </applet>
</body>
</html>

```

Fișierul Java corespunzător appletului este următorul:

```

import java.applet.*;
import java.awt.*;

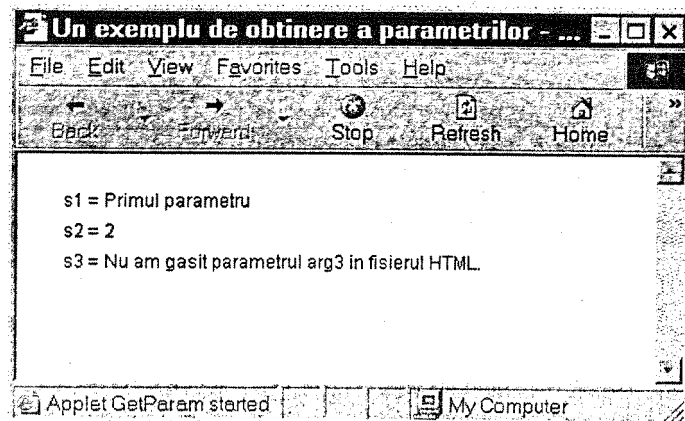
public class GetParam extends Applet {
    String s1, s2, s3;

    public void init() {
        if ((s1 = getParameter("arg1")) == null)
            s1 = "Nu am gasit parametrul arg1 in fisierul HTML.";
        if ((s2 = getParameter("arg2")) == null)
            s2 = "Nu am gasit parametrul arg2 in fisierul HTML.";
        if ((s3 = getParameter("arg3")) == null)
            s3 = "Nu am gasit parametrul arg3 in fisierul HTML.";
    }

    public void paint(Graphics g) {
        g.drawString("s1 = " + s1, 20, 20);
        g.drawString("s2 = " + s2, 20, 40);
        g.drawString("s3 = " + s3, 20, 60);
    }
}

```

În cadrul navigatorului Web, appletul va apărea astfel:



7.2.3. Contextul unui applet

Contextul unui applet este dat de interfața `AppletContext` din pachetul `java.applet`. Aceasta conține toate appleturile din același document, iar metodele sale sunt utile pentru aflarea informațiilor referitoare la mediul în care se execută appletul.

Metoda `getAppletContext()` din clasa `Applet` returnează o referință la un obiect care implementează interfața `AppletContext`. Astfel, având un applet, îi putem determina cu ușurință contextul. Cele mai importante metode pe care le oferă interfața `AppletContext` sunt:

Prototipul metodei	Descrierea metodei
<code>public Applet getApplet (String name)</code>	Returnează o referință la appletul pentru care s-a stabilit numele specificat (valoarea dată de atributul <code>name</code>). Dacă nu există appletul căutat se va întoarce <code>null</code> .
<code>public Enumeration getApplets ()</code>	Returnează o enumerare a tuturor appleturilor din pagina Web curentă.
<code>public void showDocument (URL url)</code>	Se va încărca în fereastra navigatorului documentul de la URL-ul specificat. Dacă appletul nu este rulat de un navigator, atunci apelul acestei metode este ignorat.
<code>public void showDocument (URL url, String tinta)</code>	În funcție de ținta specificată, documentul de la URL-ul specificat se va încărca în cadrul (<i>frame</i> -ul) în care se află appletul (<code>tinta="_self"</code>), în cadrul părinte (<code>tinta="_parent"</code>), în fereastra navigatorului (<code>tinta="_top"</code>), într-o fereastră nouă fără nume (<code>tinta="_blank"</code>) sau într-o fereastră cu numele dat de ținta (dacă are o valoare diferită de cele de mai sus).
<code>public void showStatus (String mesaj)</code>	Afișează mesajul specificat în bara de stare a navigatorului.

Exemplul 7.2.6. La încărcarea paginii `FCSSite.html`, appletul va redirecta navigatorul către situl Facultății de Informatică din Iași. Conținutul paginii Web este:

```

<html>
<head>
    <title>Redirectare la situl FCS</title>
</head>

<body>
    <applet code="FCSSite.class" width="10" height="10">
    </applet>
</body>
</html>

```

iar appletul va avea următorul cod:

```
import java.applet.*;
import java.net.*;

public class FCSsite extends Applet {
    public void start() {
        AppletContext context = getAppletContext();
        try {
            context.showDocument(new URL("http://www.infoiasi.ro"));
        } catch (MalformedURLException e) {
            System.out.println("ERR2");
        }
    }
}
```

7.2.4. Comunicarea dintre appleturi

Două appleturi pot comunica între ele dacă sunt de pe același server și sunt integrate în aceeași pagină Web. Modalitatea de comunicare este simplă: unul dintre appleturi obține contextul său (care este și al celui alt applet, întrucât fac parte din aceeași pagină Web) și poate obține o referință la celălalt applet prin apelul metodei `getApplet()` din context (acest lucru este posibil dacă cele două appleturi fac parte din același pachet, adică sunt în același director), după care putem apela metodele publice ale acestuia. Prin apelul de către un applet la metodele publice ale altui applet se poate realiza comunicarea.

Exemplul 7.2.7. Fișierul `comunicare.html` conține două appleturi: receptor și emitor. Appletul emitor va transmite un mesaj appletului receptor, iar acesta îl va afișa. Fișierul HTML este următorul:

```
<html>
<head>
    <title>Comunicare intre appleturi</title>
</head>

<body>
    <applet name="receptor" code="AppletReceptor.class"
        width="400" height="60">
    </applet>
    <hr>
    <applet name="emitor" code="AppletEmitator.class"
        width="400" height="60">
    </applet>
</body>
</html>
```

Appletul receptor are următorul cod:

```
import java.applet.Applet;
import java.awt.Graphics;

public class AppletReceptor extends Applet {
    private String mesaj="Nu am primit nici un mesaj!";

    public void primește(String mesaj) {
        this.mesaj = mesaj;
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString("Receptor",10,20);
        g.drawString(mesaj,10,50);
    }
}
```

Atunci când primește un mesaj, se va modifica data membră privată `mesaj`, după care se va invoca metoda `repaint()`, care la rândul ei va apela indirect metoda `paint()`. Acest lucru este util pentru afișarea mesajului primit.

Appletul emițător conține următorul cod Java:

```
import java.applet.*;
import java.awt.Graphics;

public class AppletEmitator extends Applet {
    private String msg="";

    public void start() {
        // obtinerea contextului
        AppletContext context = getAppletContext();
        int i=0;
        AppletReceptor receptor = null;
        do {
            receptor = (AppletReceptor) context.getApplet("receptor");
            i++;
            System.out.println(i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
        } while(i<1000 && receptor==null);
        if (receptor == null)
            msg = "Nu am gasit receptorul!";
        else {
```

```

        receptor.primeste("Salut prietene!");
        msg = "Mesajul a fost trimis!";
    }
    repaint();
}
public void paint(Graphics g) {
    g.drawString("Emitator", 10, 20);
    g.drawString(msg, 10, 50);
}
}

```

Mai întâi se obține contextul appletului după care se încearcă, eventual de mai multe ori, obținerea unei referințe la appletul receptor. Se încearcă de mai multe ori atunci când pagina Web nu este încărcată complet și appletul receptor încă nu este vizibil. Când s-a terminat de încărcat, atunci vom putea obține referința dorită și vom apela metoda `primeste()` a appletului receptor. Acesta va primi mesajul și-l va afișa.

În exemplul ilustrat anterior, comunicarea este unidirecțională, de la appletul emițător la cel receptor. Comunicarea se poate realiza și în ambele sensuri dacă emițătorul va conține o metodă publică de primire a mesajului, iar receptorul va trebui să emită și el mesaje. Se pot schimba mai multe mesaje (fie ele sincrone sau asincrone) între appleturi.

7.2.5. Redarea sunetelor și imaginilor

Programele Java pot avea ca ieșire elemente grafice și/sau sunete. Informațiile de grafică și sunete sunt memorate în fișiere. Cele mai cunoscute formate de fișiere grafice sunt GIF (Graphics Interchange Format), JPEG (Joint Photographic Experts Group) și PNG (Portable Network Graphics). Fișierele GIF au extensia `.gif` și folosesc pentru reprezentarea unui pixel cel mult 8 biți. Fișierele JPEG au extensia `.jpg` și utilizează pentru reprezentarea unui pixel 24 biți. JPEG este mai bun și ca rezoluție decât GIF și are o compresie superioară a informației grafice. Fișierele PNG au extensia `.png`, au 8 biți pentru fiecare pixel și oferă o mai bună compresie decât GIF-urile. Din păcate, nu toate navigatoarele Web recunosc acest format.

Este recomandată utilizarea fișierelor GIF atunci când imaginile conțin mai puțin de 256 de culori.

7.2.5.1. Afișarea conținutului grafic

Presupunem că dorim să vizualizăm o imagine grafică conținută în fișierul `pictura.gif`. Pentru acest lucru trebuie să parcurgem două etape:

1. obținerea unei referințe la imagine;
2. încărcarea și afișarea imaginii.

În prima etapă se invocă una dintre cele două forme ale metodei `getImage()` din clasa `Applet`:

- `public Image getImage(URL urlImage)`
- `public Image getImage(URL urlDirector, String numeFisier)`

unde `urlImage` este URL-ul absolut la imagine, `urlDirector` este URL-ul unui director care conține imaginea, iar `numeFisier` este numele fișierului grafic, eventual un URL relativ față de adresa directorului specificat.

Exemple:

- `Image image = getImage(getDocumentBase(), "pictura.gif");`
- `Image image = getImage(new URL("http://picturi.polirom.ro/pictura.gif"));`

În primul caz, imaginea trebuie să fie plasată în același director cu documentul HTML care conține appletul. În al doilea caz, imaginea trebuie să aibă URL-ul specificat.

Încărcarea și afișarea imaginii se realizează cu metoda `drawImage()` din clasa `Graphics`. Cele mai frecvent utilizate forme ale acestei metode sunt:

- `abstract boolean drawImage(Image image, int x, int y, Color bgColor, ImageObserver observator)`
- `abstract boolean drawImage(Image image, int x, int y, ImageObserver observator)`
- `abstract boolean drawImage(Image image, int x, int y, int latime, int inaltime, Color bgColor, ImageObserver observator)`
- `abstract boolean drawImage(Image image, int x, int y, int latime, int inaltime, ImageObserver observator)`

Parametrul `image` este o referință la imaginea care se va afișa (cea obținută la pasul 1); `x` și `y` reprezintă coordonatele unde se va afișa imaginea (respectiv colțul din stânga-sus); `latime` și `inaltime` stabilesc lățimea și respectiv înălțimea imaginii (implicite sunt dimensiunile reale ale imaginii); `bgColor` reprezintă culoarea de fundal utilizată și are efect doar dacă imaginea are porțiuni transparente; `observator` este un obiect care indică cât din imagine s-a încărcat și, de obicei, este obiectul curent, adică `this`. Metoda se termină imediat chiar dacă imaginea nu s-a încărcat complet. Se returnează `true` dacă imaginea s-a încărcat complet și `false` în caz contrar.

Atât `Image`, cât și `Graphics` sunt clase abstracte. `ImageObserver` este o interfață din pachetul `java.awt.image` care furnizează informații despre încărcarea imaginii. Clasa `Applet` implementează interfața `ImageObserver`.

Exemplu de utilizare a metodei `drawImage()`:

```

boolean b = g.drawImage(image, 20, 20, 100, 100, this);

```

Exemplul 7.2.8. Afișarea unei imagini.

```

import java.applet.Applet;
import java.awt.*;

public class InserareImagine extends Applet {
    private Image imagine;

```

```

public void init() {
    // obtinerea referintei la imagine
    imagine = getImage(getDocumentBase(), "imagine.jpg");
}

public void paint(Graphics g) {
    // desenarea imaginii
    boolean b = g.drawImage(imagine, 20, 20, 100, 100, this);
}
}

```

7.2.5.2. Redarea sunetelor

Ca și imaginile grafice, sunetele sunt memorate în fișiere. Sunt suportate următoarele formate: AIFF, AU, WAV, MIDI (tipul 0 și 1), RMF. Tipul de fișier cel mai utilizat este fișierul audio, cu extensia .au. Ca și la imagini, mai întâi trebuie să obținem o referință la un fișier audio, iar acest lucru se realizează cu metoda `getAudioClip()` din clasa `Applet`:

- `public AudioClip getAudioClip(URL urlImagine)`
- `public AudioClip getAudioClip(URL urlDirector, String numeFisier)`

unde parametrii au aceeași semnificație ca la metoda `getImage()` prezentată în secțiunea anterioară, cu deosebirea că acum ne referim la un fișier audio, iar tipul returnat este `AudioClip`.

Interfața `AudioClip` se găsește în pachetul `java.applet` și conține următoarele metode:

Prototipul metodei	Descrierea metodei
<code>public void loop()</code>	Pornește difuzarea clipului audio de mai multe ori.
<code>public void play()</code>	Difuzează conținutul audio o singură dată.
<code>public void stop()</code>	Oprește difuzarea clipului.

Programul Java continuă să se execute în timp ce sunetul este ascultat.

Exemplul 7.2.9. Appletul pune la dispoziție trei butoane, câte unul pentru fiecare metodă din clasa `AudioClip`:

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Audio extends Applet
    implements ActionListener {

    private Button play, loop, stop;

```

```

private AudioClip sound;

public void init() {
    // crearea butoanelor
    play = new Button("Play");
    add(play);
    play.addActionListener(this);
    loop = new Button("Loop");
    add(loop);
    loop.addActionListener(this);
    stop = new Button("Stop");
    add(stop);
    stop.addActionListener(this);
    // obtinerea referintei la clip
    sound = getAudioClip(getDocumentBase(), "clip.wav");
}

public void actionPerformed(ActionEvent event) {
    // in functie de butonul apasat
    // se va apela metoda corespunzatoare
    if (event.getSource() == play)
        sound.play();
    if (event.getSource() == loop)
        sound.loop();
    if (event.getSource() == stop)
        sound.stop();
}
}

```

7.2.6. Transformarea unui applet într-o aplicație de sine stătătoare

Se poate întâmpla, de multe ori, să dorim ca un applet să fie și aplicație de sine stătătoare. Pentru aceasta va trebui să adăugăm metoda publică și statică `main()` care va crea o fereastră în care se va „executa” respectivul applet. Comentariile din al doilea program Java din exemplul următor indică etapele ce trebuie urmate ca appletul să fie și aplicație de sine stătătoare.

Exemplul 7.2.10. Fie următorul applet simplu pentru care dorim să fie și aplicație de sine stătătoare în același timp:

```

import java.applet.Applet;
import java.awt.*;

public class AppletSimplu extends Applet {

```

```

    public void paint(Graphics g) {
        g.drawString("Applet simplu.", 10, 20);
    }
}

```

Appletul nu face altceva decât să afișeze un mesaj. Adăugând metoda `main()`, appletul va deveni și aplicație de sine stătătoare:

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class AppletSimplu extends Applet {
    public void paint(Graphics g) {
        g.drawString("Applet simplu.", 10, 20);
    }
    public static void main(String args[]) {
        // cream o instanta a appletului
        AppletSimplu applet = new AppletSimplu();
        // cream o fereastră
        Frame fereastră = new Frame("Applet simplu");
        // adaugam appletul la fereastră
        fereastră.add(applet, BorderLayout.CENTER);
        // stabilim dimensiunile ferestrei
        fereastră.setSize(200,70);
        // stabilim un ascultator (inchidere)
        // pentru operatiile cu fereastră
        fereastră.addWindowListener(new MyWindowListener());
        // initializam appletul
        applet.init();
        // pornim appletul
        applet.start();
        // activam fereastră
        fereastră.show();
        // distrugem appletul
        applet.destroy();
    }
}

/* Aceasta clasa este responsabila cu inchiderea ferestrei
 * atunci cand s-a apasat butonul Close al ferestrei
 */
class MyWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        Window win = e.getWindow();

```

```

        win.dispose();
    }
}

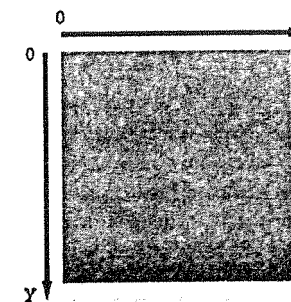
```

În aplicația Java de mai sus, a trebuit să mai declarăm o clasă care să se ocupe de închiderea ferestrei. Informații amănunțite privitoare la lucrul cu ferestre se găsesc în secțiunea 7.6.

7.3. Grafica în Java

Biblioteca AWT (eng. *Abstract Window Toolkit*) conține toate clasele pentru crearea interfețelor cu utilizatorul și pentru desenarea graficelor sau imaginilor.

Clasa abstractă `Graphics` face parte din pachetul `java.awt` și se referă la ecranul grafic Java. Grafica Java se bazează pe pixeli și pe un sistem ortogonal de axe. Astfel, avem axa Ox pe orizontală și Oy pe verticală. Punctul de coordonate $(0,0)$ se află în colțul din stânga-sus al suprafeței grafice.



Axele de coordonate în Java

De obicei, dimensiunea unei suprafețe grafice pentru un applet este de 300 pixeli lungime (pe Ox) și 200 pixeli înălțime (pe Oy).

Cele mai utilizate metode ale clasei `Graphics` sunt explicate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
<pre>public abstract void drawString (String str, int x, int y)</pre>	Desenează șirul de caractere <code>str</code> , începând cu poziția (x, y) . Se utilizează fontul și culoarea curentă.
<pre>public abstract void drawLine (int x1, int y1, int x2, int y2)</pre>	Desenează o linie între punctele de coordonate $(x1, y1)$ și $(x2, y2)$ utilizând culoarea curentă.
<pre>public abstract void drawRect (int x)</pre>	Desenează chenarul unui dreptunghi având colțul din stânga-sus la coordonata (x, y) , iar cel din dreapta-jos la $(x+lungime, y+inaltime)$.

Prototipul metodei	Descrierea metodei
public abstract void drawOval (int x, int y, int lungime, int inaltime)	Desenează chenarul unei elipse incluse în dreptunghiul dat de parametrii metodei (care au aceeași semnificație ca cei ai metodei drawRect ()).
public abstract void drawArc (int x, int y, int lungime, int inaltime, int unghiInceput, int unghiSfarsit)	Desenează un arc de elipsă încadrat într-un dreptunghi dat de primii patru parametri. unghiInceput este unghiul de început, iar unghiSfarsit este unghiul cu care se termină arcul și este relativ la cel de început. Măsura unghiurilor este dată în grade hexazecimale.

Funcțiile de „umplere” sunt **fillRect**(), **fillArc**(), **fillOval**() și au aceeași parametri ca funcțiile echivalente de desenare (de tip **draw...**), diferența constând în colorarea interiorului utilizând culoarea curentă.

Pentru a lucra cu altă culoare în afară de cea neagră, va trebui să utilizăm obiecte din clasa **java.awt.Color**. Aceasta pune la dispoziție pentru culorile des folosite mai multe date membre statice de tip **Color**. Acestea sunt: **black** (negru), **blue** (albastru), **cyan**, **darkGray** (gri închis), **gray** (gri), **green** (verde), **lightGray** (gri deschis), **magenta**, **orange** (oran), **pink** (roz), **red** (roșu), **white** (alb) și **yellow** (galben). Pentru a obține o altă culoare, vom utiliza constructorul:

```
public Color(int r, int g, int b);
```

care are ca parametri codul RGB (Red-Green-Blue). Aceștia au valori cuprinse între 0 și 255 și indică gradul în care se combină cele trei culori (roșu, verde și respectiv albastru) pentru obținerea noii culori. Dacă dorim să obținem o culoare care este transparentă într-o anumită proporție, atunci vom utiliza constructorul:

```
public Color(int r, int g, int b, int transparenta);
```

unde primii trei parametri formează codul RGB al culorii, iar ultimul are valoarea cuprinsă tot între 0 și 255, 0 desemnând o transparență totală, iar 255 stabilind o culoare opacă.

Pentru stabilirea culorii de fundal a unui applet putem apela metoda **setBackground()** moștenită de **Applet** din clasa **Component**, iar culoarea pentru desenare, cu metoda **setColor()** din clasa **Graphics**. Ambele metode au un singur parametru de tip **Color**.

Exemplul 7.3.1. Appletul va utiliza metodele explicate anterior. Se vor evidenția dreptunghiurile circumscrise arcelor.

```
import java.awt.*;
import java.applet.Applet;

public class DesenSimplu extends Applet {
    public void paint(Graphics g) {
        g.drawString("Exercitiu de grafica", 10, 15);
```

```
g.drawRect(30, 30, 80, 40);
g.drawOval(120, 30, 50, 50);
g.setColor(Color.red);
g.fillRect(30, 100, 80, 40);
g.fillOval(120, 100, 50, 50);
g.drawLine(30, 160, 130, 170);
g.drawArc(30, 180, 30, 50, 60, 90);
g.fillArc(120, 180, 50, 50, 60, 40);
g.setColor(Color.black);
g.drawRect(30, 180, 30, 50);
g.drawRect(120, 180, 50, 50);
setBackground(Color.lightGray);
}
```

Exemplul 7.3.2. Appletul va desena două triunghiuri isoscele și va calcula aria acestora.

```
import java.awt.*;
import java.applet.Applet;

public class TriunghiIsoscel extends Applet {
    public void paint(Graphics g) {
        g.setColor(Color.blue);
        deseneaza_triunghi(g, 80, 200, 100, 110);
        g.drawString("Aria triunghiului este " +
            aria_triunghi(100, 110), 135, 130);
        g.setColor(Color.magenta);
        deseneaza_triunghi(g, 125, 220, 60, 70);
        g.drawString("Aria triunghiului este " +
            aria_triunghi(60, 70), 150, 180);
    }

    private void deseneaza_triunghi(Graphics g, int bottomX,
        int bottomY, int baza, int inaltime) {
        int rightX = bottomX + baza;
        int topX = bottomX + baza/2;
        int topY = bottomY - inaltime;
        g.drawLine(bottomX, bottomY, rightX, bottomY);
        g.drawLine(rightX, bottomY, topX, topY);
        g.drawLine(topX, topY, bottomX, bottomY);
    }

    private int aria_triunghi(int baza, int inaltime) {
        int aria = (baza * inaltime)/2;
```

```

    return aria;
}
}

```

Observație! Toate metodele private (definite de noi) sunt declarate și apelate, în schimb funcția `paint()` este doar declarată, și nu apelată!!! Navigatorul Web sau appletviewer-ul inițializează appletul și apelează funcția `paint()`. De aceea, aceasta nu poate fi private, ci public.

Exemplul 7.3.3. Următorul applet animează o minge pe ecran. Mingea se află într-un dreptunghi răsfrând din pereții acestuia.

```

import java.awt.*;
import java.applet.Applet;

public class Minge extends Applet {
    private int x = 7, xChange = 7;
    private int y = 2, yChange = 2;
    private int diameter = 10;
    private int rectLeftX = 0, rectRightX = 100;
    private int rectTopY = 0, rectBottomY = 100;
    private boolean ok;

    // initializarea appletului
    public void init() {
        ok = true;
    }

    public void paint(Graphics g) {
        // se deseneaza dreptunghiul
        g.drawRect(rectLeftX, rectTopY, rectRightX - rectLeftX + 10,
            rectBottomY - rectTopY + 10);

        while (ok) {
            // se obtine culoarea fundalului
            Color culoareFond = getBackground();
            // culoarea fundalului devine culoarea de desenare
            // (practic se sterge figura)
            g.setColor(culoareFond);
            // se sterge mingea pentru a fi desenata la noua pozitie
            g.fillOval(x, y, diameter, diameter);
            // daca se intalneste peretele se schimba directia de
            // miscare
            if (x + xChange <= rectLeftX)    xChange = - xChange;
            if (x + xChange >= rectRightX)   xChange = - xChange;
            if (y + yChange <= rectTopY)     yChange = - yChange;

```

```

        if (y + yChange >= rectBottomY) yChange = - yChange;
        // se modifica noua pozitie a mingii
        x = x + xChange;
        y = y + yChange;
        // se stabileste culoarea rosie pentru desenare
        g.setColor(Color.red);
        // se deseneaza mingea
        g.fillOval(x, y, diameter, diameter);
        try {
            Thread.sleep(40);
        } catch (InterruptedException e) { }
    }

    // se apeleaza la distrugerea appletului
    // si are ca efect terminarea metodei paint()
    public void destroy() {
        ok = false;
    }
}

```

7.4. Componente și evenimente

O *componentă* (grafică) este o entitate grafică care poate fi adăugată la suprafața de afișare a unui program sau applet Java. Utilizatorul poate interacționa cu programul prin intermediul acestor componente. La acționarea asupra unei astfel de componente se emite un eveniment. Pentru ca la apariția unui eveniment să-i dăm programului o anumită funcționalitate, va trebuie să atașăm un „ascultător” care să implementeze efectul dorit.

Un *eveniment* este o acțiune impusă de către utilizator. De exemplu, acțiuni sunt apăsarea unei taste sau a unui buton de mouse de către utilizator, manevrarea unei bare de derulare (*Scroll*), marcarea unei opțiuni dintr-un meniu.

Așadar, în loc să ne imaginăm un program ca un șir de instrucțiuni, putem să ne închipuim secțiunile programului ca fiind activate de un eveniment (se mai spune că programul este „condus de evenimente”).

Un program „condus de evenimente” (eng. *event-driven*) trebuie să rezolve problema evenimentelor (eng. *event-handling*). Detectarea evenimentelor și rezolvarea sarcinii cerute se numește *rezolvarea evenimentului*. Programele care utilizează acest stil de interfață cu utilizatorul se numesc „conduse de evenimente”.

7.4.1. Bucla evenimentelor

Nu trebuie să codificăm bucla evenimentelor și nu trebuie să facem recunoașterea evenimentelor la nivelul de jos. În Java, creăm o secțiune de programe (metode) pe care sistemul Java le apelează automat când un eveniment are loc, în plus dându-le detalii despre eveniment. Apoi, aceste detalii (de exemplu, apăsarea butonului stânga de la mouse) sunt utile pentru a asigura că procesul cerut are loc.

În Java, evenimentele sunt clasificate. De exemplu, barele de derulare (eng. *scrollbars*) sunt, de obicei, utilizate pentru modificarea valorii sau vizualizarea unei alte părți a ecranului. Butoanele, în schimb, prin apăsarea lor, sunt folosite pentru inițierea sau terminarea unei activități pentru îndeplinirea unei cerințe. Obiectele (interfețele) care se ocupă de aceste lucruri se numesc ascultători (eng. *listeners*), au sufixul *Listener*, pentru butoane există clasa *ActionListener*, iar pentru barele de defilare, *AdjustmentListener*.

Fiecare ascultător conține câte o metodă pentru fiecare tip de acțiune care ne interesează (o metodă pentru derularea barei de defilare, o metodă pentru acționarea unui click etc.). În plus, trebuie să desemnăm un ascultător pentru componenta (eng. *scrollbar*) care produce evenimente, iar metodele acestuia vor fi apelate automat când apare evenimentul corespunzător. Prin redefinirea metodelor ascultătorului putem da componentei noastre funcționalitatea dorită.

Fiecărui eveniment îi corespunde o clasă care are sufixul *Event*. Ascultătorii și evenimentele sunt clase care fac parte din pachetul *java.awt.event*. Tot în acest pachet se găsesc unele clase care implementează interfețele ascultătorilor și nu au nici un efect la primirea evenimentelor. Aceste clase se mai numesc adaptori (eng. *adapters*).

Exemplul 7.4.1. Programul Java de mai jos (condus de evenimente) afișează numărul corespunzător cu deplasarea unei bare de defilare.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Slider extends Applet
    implements AdjustmentListener {

    private Scrollbar slider;
    private int sliderValue = 0;

    public void init() {
        // se creeaza o bara de derulare
        slider = new Scrollbar(Scrollbar.HORIZONTAL,
            0, 1, 0, 100);
        // se adauga bara la suprafata de desenare
        // a appletului
```

```
        add(slider);
        // se stabileste un ascultator pentru bara
        slider.addAdjustmentListener(this);
    }

    public void paint(Graphics g) {
        g.drawString("Valoarea curenta este " +
            sliderValue, 100, 70);
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        // se salveaza pozitia barei de defilare
        sliderValue = slider.getValue();
        // se invoca redesenarea appletului
        repaint();
    }
}
```

Clasa *Slider* implementează interfața *AdjustmentListener*, care este un ascultător pentru evenimentele specifice barelor de derulare. În metoda de inițializare se creează o bară de derulare care este adăugată la suprafața de desenare a appletului. La finalul metodei se stabilește un ascultător pentru bara de defilare prin apelul metodei *addAdjustmentListener()*. Aceasta are ca parametru un obiect care implementează interfața *AdjustmentListener*. Aceasta posedă metoda *adjustmentValueChanged()* care este apelată ori de câte ori bara de defilare este utilizată.

Acest program permite doar manipularea barei de defilare. De fiecare dată când mișcăm bara de defilare, metoda *adjustmentValueChanged()* este invocată. În cazul nostru, metoda *adjustmentValueChanged()* copiază noua valoare din bara de defilare în variabila *sliderValue*, apoi redesenează suprafața appletului.

Metoda *void adjustmentValueChanged(AdjustmentEvent e)* este utilizată pentru procesarea evenimentelor pentru bara de defilare. Când un astfel de eveniment apare, această metodă este automat apelată.

Pentru crearea unei bare de derulare s-a instanțiat clasa *java.awt.Scrollbar* apelându-se la constructorul:

```
public Scrollbar(int orientare, int valoare, int vizibilitate,
    int minim, int maxim)
```

Parametrul *orientare* indică orientarea barei de defilare: orizontală (caz în care se utilizează data membră statică *Scrollbar.HORIZONTAL*) sau verticală (*Scrollbar.VERTICAL*). Al doilea parametru indică valoarea inițială pe care o va avea bara, iar al treilea se referă la dimensiunea dreptunghiului din bara de derulare a cărui glisare duce la modificarea valorii barei. Parametrii *minim* și *maxim* se referă la marginile domeniului din care ia valori deplasamentul barei de defilare.

Exemplul 7.4.2. Programul afișează o bară de defilare verticală, iar o dată cu modificarea sa se schimbă și nivelul de umplere a unui dreptunghi.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class VSlider extends Applet
    implements AdjustmentListener {
    private Scrollbar slider;
    private int sliderValue;

    public void init() {
        // crearea unei bare de derulare verticale
        slider = new Scrollbar(Scrollbar.VERTICAL,
            0, 1, 0, 100);
        // adaugarea barei la applet
        add(slider);
        // stabilirea unui ascultator pentru bara
        slider.addAdjustmentListener(this);
    }

    public void paint(Graphics g) {
        // se afiseaza in bara de stare a navigatorului
        showStatus("Valoarea barei Scroll este " +
            sliderValue);
        // desenarea dreptunghiului
        g.drawRect(40, 80, 60, 100);
        // umplerea acestuia in functie de deplasarea
        // barei de derulare
        g.fillRect(40, 80, 60, sliderValue);
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        sliderValue = slider.getValue();
        repaint();
    }
}
```

Acest exemplu este similar cu exemplul 7.4.1. Observăm în plus că, la desenare, metoda `fillRect()` umple un dreptunghi care are înălțimea egală cu valoarea lui `sliderValue`, adică cu valoarea derulării barei verticale.

Programul este condus de evenimente, deoarece așteaptă utilizarea barei de defilare. Evenimentele sunt procesate de metoda `adjustmentValueChanged()`. Aceasta are un singur parametru de tip `AdjustmentEvent` care conține detalii despre eveniment.

În programul nostru există doar un eveniment (mișcarea barei de defilare).

O bară de derulare poate avea doar valori întregi în domeniul stabilit. Se poate întâmpla să avem nevoie de valori care să fie numere zecimale, de exemplu 0.0, 1.0, 2.0... Pentru aceasta vom scala bara de defilare.

Vom considera un exemplu de conversie a inch-ilor (din domeniul {0, ..., 10}) la centimetri, unde 1 inch = 2.54 cm. Putem seta bara de defilare la domeniul 0..10, dar pentru mai multă precizie vom crea bara de defilare în domeniul {0, ..., 100} și vom scala acești întregi către reali rezultând pași de 0.1 inch.

Exemplul 7.4.3. Conversia din inch în centimetri.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Inch2Centimetri extends Applet
    implements AdjustmentListener {

    private Scrollbar slider;
    private float sliderValue;

    public void init() {
        slider = new Scrollbar(Scrollbar.HORIZONTAL,
            0, 1, 0, 100);
        add(slider);
        slider.addAdjustmentListener(this);
    }

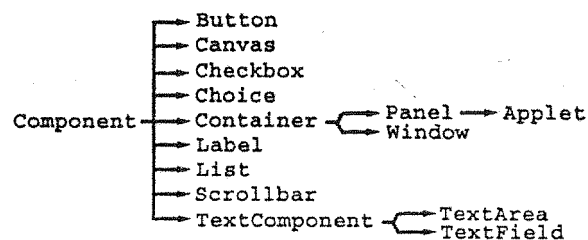
    public void paint(Graphics g) {
        // transformarea din inch in centimetri
        float cm = sliderValue * 2.54f;
        g.drawString("Inci= " + sliderValue +
            " Cm=" + cm, 100, 100);
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        // scalarea din numere intregi in numere zecimale
        sliderValue = (float) slider.getValue() / 10;
        repaint();
    }
}
```

7.4.2. Componente

O componentă este un obiect care are reprezentare grafică, poate fi afișat pe ecran și poate interacționa cu utilizatorul.

Toate componentele, mai puțin meniurile, sunt derivate direct sau indirect din clasa `Component`. Metodele acestei clase se vor regăsi și în clasele derivate și vor stabili proprietățile generale (respectiv obținerea acestora) ale unei componente. Există componente care includ alte componente.



Ierarhia componentelor în AWT

Metodele cele mai semnificative din clasa `Component` sunt următoarele:

Prototipul metodei	Descrierea metodei
<code>public Color getBackground()</code>	Obține culoarea de fundal.
<code>public void setBackground (Color c)</code>	Stabilește culoarea fundalului.
<code>public Color getForeground()</code>	Întoarce culoarea de scriere sau de desenare.
<code>public void setForeground (Color c)</code>	Setează culoarea de scriere sau de desenare.
<code>public Component getComponentAt (int x, int y)</code>	Întoarce componenta de la coordonatele (x, y) dacă aceasta este componenta curentă sau sub-componenta acestuia și null în caz contrar.
<code>public int getHeight()</code>	Obține înălțimea componentei.
<code>public int getWidth()</code>	Obține lățimea componentei.
<code>public void setSize (int latime, int inaltime)</code>	Stabilește lățimea și respectiv înălțimea componentei.
<code>public void setBounds (int x, int y, int latime, int inaltime)</code>	Stabilește poziția și dimensiunile componentei (colțul din stânga sus va fi la coordonata (x, y)).
<code>public int getX()</code> <code>public int getY()</code>	Cele două metode determină coordonata la care se află componenta.
<code>public String getName()</code>	Returnează numele componentei.
<code>public void setName (String nume)</code>	Stabilește numele componentei.

Prototipul metodei	Descrierea metodei
<code>public Component getParent()</code>	Întoarce componenta părinte.
<code>public boolean hasFocus()</code>	Testează dacă componenta este activă.
<code>public boolean isShowing()</code>	Testează dacă componenta este vizibilă pe ecran.
<code>public boolean isEnabled()</code>	Testează dacă componenta este disponibilă.
<code>public void setEnable (boolean b)</code>	Stabilește dacă componenta este disponibilă.
<code>public void paint (Graphics g)</code>	Desenează componenta. g este contextul grafic pentru desenare.
<code>public void paintAll (Graphics g)</code>	Desenează componenta și subcomponentele sale.
<code>public void repaint()</code>	Redesenează componenta.

O importanță aparte o au componentele care conțin alte componente. Acestea sunt denumite containere.

7.4.3. Containere

Un container este o componentă care poate conține alte componente. Unui container îi corespunde clasa `java.awt.Container` care este derivată din clasa `Component`.

Clasa `Container` are doar un constructor implicit, iar metodele cele mai importante sunt explicate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
<code>public Component add (Component c)</code>	Adaugă componenta specificată la sfârșitul containerului. Returnează componenta adăugată.
<code>public Component add (Component c, int index)</code>	Adaugă componenta specificată în container pe poziția indicată. Returnează componenta adăugată.
<code>public void add (Component c, Object constrangeri)</code>	Adaugă componenta specificată la sfârșitul containerului și stabilește constrângeri pentru poziționare. A se vedea subcapitolul 7.5.
<code>public void add (Component c, Object constrangeri, int index)</code>	Adaugă componenta specificată în container pe poziția indicată și stabilește constrângeri pentru poziționare. A se vedea subcapitolul 7.5.
<code>public Component findComponentAt (int x, int y)</code>	Returnează subcomponenta care se află la coordonata indicată sau null în caz de eșec.
<code>public int getComponentCount()</code>	Returnează numărul de componente din container.
<code>public Component getComponent (int n)</code>	Întoarce componenta de pe poziția n din container.

Prototipul metodei	Descrierea metodei
public Component[] getComponents()	Returnează un tablou cu toate componentele din container.
public void remove (Component c)	Elimină componenta specificată.
public void remove (int index)	Elimină componenta cu indexul indicat.
public void removeAll()	Elimină toate componentele din container.

Clasele cele mai importante care sunt derivate din clasa Container sunt Panel și Window care vor fi prezentate ulterior în acest capitol.

7.4.4. Etichete

În continuare, vom vedea cum putem scrie un text lângă o bară de defilare (bineînțeles, fără a preciza coordonatele). Vom utiliza funcția `add()` pentru a plasa o nouă componentă pe ecran. Implicit, se poziționează componentele centrat, de la stânga la dreapta.

Vom introduce componenta Label care conține text și care se adaugă la fel ca barele de defilare.

Exemplul 7.4.4. Inserarea unei etichete într-un applet.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class AppletCuEtichete extends Applet
    implements AdjustmentListener {

    private Scrollbar bar1, bar2;
    private int bar1Value = 0;
    private int bar2Value = 0;

    public void init() {
        Label titlu1, titlu2;
        // crearea unei etichete
        titlu1 = new Label("Sus:");
        // adaugarea etichetei la applet
        add(titlu1);
        bar1 = new Scrollbar(Scrollbar.HORIZONTAL,
            0, 1, 0, 100);
        add(bar1);
        bar1.addAdjustmentListener(this);
        // crearea altei etichete
        titlu2 = new Label("Jos:");
```

```
// adaugarea etichetei la applet
add(titlu2);
bar2 = new Scrollbar(Scrollbar.HORIZONTAL,
    0, 1, 0, 100);
add(bar2);
bar2.addAdjustmentListener(this);
}

public void paint(Graphics g) {
    g.drawString("Valoarea de sus este " +
        bar1Value, 100, 100);
    g.drawString("Valoarea de jos este " +
        bar2Value, 100, 150);
}

public void adjustmentValueChanged(AdjustmentEvent e) {
    bar1Value = bar1.getValue();
    bar2Value = bar2.getValue();
    repaint();
}
}
```

O etichetă nu este activă (în sensul că acționarea unui click asupra sa nu produce evenimente) și nu trebuie să ne referim la o etichetă mai târziu. De aceea, domeniul unei etichete poate fi local în metoda de inițializare.

Mai observăm că cele două bare de defilare au același ascultător, iar dacă s-a modificat starea uneia dintre ele, va duce la apelarea aceleiași metode. Eventual, se poate defini pentru fiecare bară câte un ascultător.

Recomandări generale:

- declarați toate datele membre ca fiind private;
- asigurați-vă că programul importă pachetele necesare;
- înregistrați câte un ascultător pentru fiecare bară;
- creați ascultătorul (ascultătorii) necesar(i).

7.4.5. Butoane

Unui buton îi corespunde un obiect din clasa `java.awt.Button`. Aceasta are doi constructori: unul implicit și altul cu un parametru de tip `String` care stabilește textul ce va apărea pe buton. Aflarea și setarea etichetei se realizează cu metodele publice `String getLabel()`, respectiv `void setLabel(String eticheta)`.

Ascultătorul unui buton trebuie să implementeze interfața `ActionListener`. Aceasta are o singură metodă care este apelată atunci când s-a apăsat butonul:

```
public void actionPerformed(ActionEvent e)
```

Adăugarea unui ascultător la un buton se realizează cu metoda `addActionListener()` din clasa `Button`.

În biblioteca (clasa) `Math` există metoda statică `random()` care returnează un număr aleator de tip `double` cuprins în intervalul $[0, 1)$.

Exemplul 7.4.5. Programul permite aruncarea a două zaruri și testează dacă numerele obținute sunt egale. Pentru o nouă aruncare a zarurilor, se va apăsa pe un buton.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Zaruri extends Applet implements
    ActionListener {

    private Button aruncaZar;
    private boolean arunca = false;

    public void init() {
        // crearea unui buton
        aruncaZar = new Button("Arunca");
        // adaugarea butonului la applet
        add(aruncaZar);
        // stabilirea ascultătorului
        aruncaZar.addActionListener(this);
    }

    // metoda este apelată la apăsarea butonului
    public void actionPerformed(ActionEvent eveniment) {
        arunca = true;
        repaint();
    }

    public void paint(Graphics g) {
        int zar1, zar2;
        if (arunca) {
            zar1 = (int)(Math.random() * 6) + 1;
            zar2 = (int)(Math.random() * 6) + 1;
            g.drawString("Zarurile sunt " + zar1 + " si " +
                zar2, 20, 40);
            if (zar1 == zar2)
                g.drawString("Am castigat ! Zarurile sunt egale",
                    20, 60);
            else
```

```
g.drawString("Am pierdut ! Zarurile nu sunt egale",
    20, 60);
    }
}
```

O altă modalitate de a obține numere aleatoare este de a utiliza clasa `java.util.Random`. Aceasta posedă următoarele metode importante:

Prototipul metodei	Descrierea metodei
<code>public float nextFloat()</code>	Returnează următorul număr aleator distribuit uniform în intervalul $[0, 1)$ (de tip <code>float</code>).
<code>public double nextDouble()</code>	Returnează următorul număr aleator distribuit uniform în intervalul $[0, 1)$ (de tip <code>double</code>).
<code>public int nextInt()</code>	Returnează următorul număr de tip <code>int</code> . Toate numerele posibile de tip <code>int</code> sunt echiprobabile.
<code>public long nextLong()</code>	Returnează următorul număr de tip <code>long</code> . Toate numerele posibile de tip <code>long</code> sunt echiprobabile.
<code>public void setSeed(long seed)</code>	Setează numărul care stă la baza generatorului de numere aleatorii.

Exemplul 7.4.6. O altă modalitate de a genera numere aleatorii.

```
Random r = new Random();
float f = r.nextFloat();
g.drawString("Numarul aleator este: "+f, 50, 50);
```

7.4.6. Câmpuri și arii text

Citirea textelor în AWT se realizează prin intermediul câmpurilor text și a ariilor text. Un câmp conține o singură linie, pe când o arie text poate conține mai multe linii.

7.4.6.1. Clasa `java.awt.TextComponent`

Câmpurile text sunt definite de clasa `java.awt.TextField`, iar ariile text de `java.awt.TextArea`. Ambele sunt derivate din clasa `java.awt.TextComponent`. Aceasta posedă următoarele metode importante:

Prototipul metodei	Descrierea metodei
<code>public void addTextListener(TextListener ascultator)</code>	Atașează un ascultător de tip <code>TextListener</code> . După acest tabel este prezentată interfața <code>TextListener</code> .
<code>public Color getBackground()</code>	Obține culoarea de fundal a componentei text.

Prototipul metodei	Descrierea metodei
public void setBackground (Color c)	Stabilește culoarea de fundal a componentei.
public int getCaretPosition ()	Returnează poziția cursorului în cadrul textului.
public void setCaretPosition (int pozitie)	Mută cursorul la poziția specificată.
public boolean isEditable ()	Testează dacă textul din componentă poate fi modificat (returnează true în caz afirmativ).
public void setEditable (boolean b)	Stabilește dacă textul poate fi modificat (caz în care b are valoare true) sau nu.
public void select (int inceput, int sfarsit)	Selectează textul începând cu poziția inceput și până la poziția sfarsit.
public void selectAll ()	Selectează tot textul din componentă.
public int getSelectionStart ()	Întoarce poziția de început a textului selectat.
public int getSelectionEnd ()	Întoarce poziția de sfârșit a textului selectat.
public void setSelectionStart (int poz)	Stabilește poziția de început a selecției.
public void setSelectionEnd (int poz)	Stabilește poziția de sfârșit a selecției.
public String getSelectionText ()	Returnează textul selectat.
public String getText ()	Returnează întregul text din componentă.
public void setText ()	Stabilește textul din componentă.

Unei componente text îi putem atașa un ascultător de tip `TextListener`. Această interfață posedă o singură metodă care este apelată atunci când se modifică textul din componentă:

- `public void textValueChanged(TextEvent e)`

7.4.6.2. Clasa `java.awt.TextField`

Clasa `TextField` moștenește clasa `TextComponent` și are următorii constructori (pe lângă cel implicit):

- `public TextField(int nrColoane)`
- `public TextField(String textInitial)`
- `public TextField(String textInitial, int nrColoane)`

unde parametrul `nrColoane` se referă la lățimea componentei (exprimată în numărul de caractere), iar `textInitial` va fi textul care va apărea în cadrul câmpului.

Pentru a capta acțiunile utilizatorului cu privire la introducerea de date într-un câmp text, trebuie să asociem câmpului un ascultător de tip `ActionListener` sau `TextListener`.

În `java.lang.Integer` există funcția

- `public static int parseInt(String s)`

care convertește șirul `s` de cifre zecimale într-un întreg. Se generează o excepție de tipul `NumberFormatException`, dacă șirul nu conține caractere valide.

Exemplul 7.4.7. Programul Java conține un câmp text pentru citirea vârstei și afișează un mesaj de permisiune de vot.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Varsta extends Applet
    implements ActionListener {
    // declarăm câmpul text
    private TextField campVarsta;
    // declarăm un float care să țină vârsta
    private float varsta;

    public void init() {
        // se creează efectiv câmpul text
        campVarsta = new TextField(10);
        // se adaugă la applet
        add(campVarsta);
        // se stabilește ascultătorul
        campVarsta.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        // se încearcă convertirea textului într-o
        // valoare de tip float
        try {
            Float varstaF = Float.valueOf(campVarsta.getText());
            varsta = varstaF.floatValue();
        } catch (NumberFormatException e) {
            // în caz de eșec se afișează un mesaj la consolă
            System.err.println("Excepție de citire float");
        }
        // se invocă redesenarea suprafeței grafice pentru
        // a putea vedea schimbările survenite
        repaint();
    }

    public void paint(Graphics g) {
        // se afișează vârsta
        g.drawString("Varsta este " + varsta, 50, 50);
        // în funcție de vârsta se afișează mesajul
        // corespunzător cu privire la dreptul de vot
    }
}
```

```

    if (varsta >= 18.0f)
        g.drawString("Puteti vota", 50, 100);
    else
        g.drawString("Nu puteti vota", 50, 100);
}

```

7.4.6.3. Clasa java.awt.TextArea

Clasa `TextArea` moștenește clasa `TextComponent` (care a fost descrisă în 7.4.6.1.). Putem utiliza următorii constructori (în afara constructorului implicit):

- `public TextArea(int nrRanduri, int nrColoane)`
- `public TextArea(String textInitial)`
- `public TextArea(String textInitial, int nrRanduri, int nrColoane)`
- `public TextArea(String textInitial, int nrRanduri, int nrColoane, int bareDefilare)`

unde parametrii `nrRanduri` și `nrColoane` se referă la numărul de rânduri, respectiv de coloane vizibile, `textInitial` va desemna textul care va fi în aria text (implicit este șirul vid), iar `bareDefilare` va specifica care bare de derulare vor fi disponibile. Pentru `bareDefilare` clasa `TextArea` pune la dispoziție următoarele date membre:

- `public static int SCROLLBARS_BOTH`
ambele bare de defilare vor fi disponibile;
- `public static int SCROLLBARS_HORIZONTAL_ONLY`
doar bara de defilare orizontală va fi disponibilă;
- `public static int SCROLLBARS_VERTICAL_ONLY`
doar bara de defilare verticală va fi disponibilă;
- `public static int SCROLLBARS_NONE`
nici o bară de defilare nu va fi disponibilă.

Toți constructorii pot arunca excepția `HeadlessException`.

Cele mai importante metode din clasa `TextArea` sunt următoarele:

Prototipul metodei	Descrierea metodei
<code>public void append(String text)</code>	Adaugă textul indicat la sfârșitul textului din componentă.
<code>public void insert(String text, int pozitie)</code>	Inserează textul specificat la poziția indicată.
<code>public void replaceRange (String text, int inceput, int sfarsit)</code>	Înlocuiește textul cuprins între pozițiile <code>inceput</code> și <code>sfarsit</code> cu textul specificat.
<code>public int getColumns()</code>	Întoarce numărul de coloane ale componentei.
<code>public int getRows()</code>	Întoarce numărul de rânduri ale componentei.

Prototipul metodei	Descrierea metodei
<code>public void setColumns(int nrColoane)</code>	Restabilește numărul de coloane ale componentei.
<code>public void setRows(int nrRanduri)</code>	Restabilește numărul de rânduri ale componentei.

Exemplul 7.4.8. Appletul va conține o arie de text și două butoane. Primul va duce la afișarea textului selectat, a poziției de început și respectiv de sfârșit a selecției, iar celălalt buton va afișa întregul text. Mesajele vor apărea în bara de stare a navigatorului.

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Editor extends Applet
    implements ActionListener {
    private TextArea text;
    private Button marcaj, afisareText;

    public void init() {
        // crearea unei arii de text cu
        // 10 randuri si 20 de coloane
        text = new TextArea(10, 20);
        // adaugarea ariei text la applet
        add(text);

        // crearea unui buton
        marcaj = new Button("Afisare Text Selectat");
        add(marcaj);
        marcaj.addActionListener(this);

        afisareText = new Button("Afisare Text");
        add(afisareText);
        afisareText.addActionListener(this);
    }

    // metoda de tratare a evenimentelor aparute
    // in urma apasarii celor doua butoane
    public void actionPerformed(ActionEvent e) {
        // in functie de butonul apasat
        if (e.getSource() == marcaj) {
            // se obtine textul selectat
            String textSelectat = text.getSelectedText();
            // se obtine pozitia de start a selectiei
            int initial = text.getSelectionStart();

```

```

        // se obtine pozitia de sfarsit a selectiei
        int ultim = text.getSelectionEnd();
        // se formeaza mesajul care va fi afisat in bara de
        // stare String sir = "Textul selectat incepe la pozitia " +
            initial + " pana la pozitia " + ultim +
            " si acesta este: \"" + textSelectat + "\"";
        showStatus(sir);
    }
    if (e.getSource() == afisareText) {
        // se obtine intregul text
        String textul = text.getText();
        String sir = "Textul scris este: \"" + textul + "\"";
        showStatus(sir);
    }
}

```

Exemplul 7.4.9. Următorul applet trasează graficul funcției $f(x) = a * \sin(x) + b$. Parametrii a și b iau valori în intervalul $[-5, 5]$ și sunt cu o singură zecimală.

```

import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Grafic extends Applet implements Adjustment
Listener {
    // obiectul grafic se va ocupa de desenarea propriu-zisa
    // a graficului functiei
    private ReprezentareGrafica grafic;
    // valorile fiecarui parametru vor fi date de starea
    // unei bare de defilare
    private Scrollbar aScrollbar, bScrollbar;

    public void init() {
        // se va instantia graficul
        grafic = new ReprezentareGrafica();

        // se va crea o eticheta pentru parametrul a
        Label aLabel = new Label("a:");
        add(aLabel);
        // se va crea o bara de defilare pentru parametrul a
        aScrollbar = new Scrollbar(Scrollbar.HORIZONTAL, 50, 1,
            0, 100);
        add(aScrollbar);
    }
}

```

```

aScrollbar.addAdjustmentListener(this);

// se va crea o eticheta pentru parametrul b
Label bLabel = new Label("b:");
add(bLabel);
// se va crea o bara de defilare pentru parametrul b
bScrollbar = new Scrollbar(Scrollbar.HORIZONTAL, 50, 1,
    0, 100);
add(bScrollbar);
bScrollbar.addAdjustmentListener(this);
}

// functia de tratare a evenimentelor
// care apar la barele de defilare
public void adjustmentValueChanged(AdjustmentEvent ev) {
    int aVal = aScrollbar.getValue();
    int bVal = bScrollbar.getValue();
    grafic.seteazaParametri(aVal, bVal);
    repaint();
}

public void paint(Graphics g) {
    grafic.deseneaza(g);
}

// clasa ReprezentareGrafica este responsabila cu
// desenarea graficului functiei
class ReprezentareGrafica {
    private final int xPixelStart = 30, xPixelEnd = 380, xOrigine =
        205;
    private final int yPixelStart = 30, yPixelEnd = 380, yOrigine =
        205;
    private final double xStart = -10d, xEnd = 10d;
    private final double scala =
        (xPixelEnd - xPixelStart) / (xEnd - xStart);
    private double a, b;
    private double aMin = -5d, aMax = 5d;
    private double bMin = -5d, bMax = 5d;

    // functia noastra
    private double functie(double x, double a, double b) {
        return a*x*Math.sin(x)+b;
    }
}

```

```

// scalarea pe horizontala
private double scalaX(int xPixel) {
    double valoare = (xPixel - xOrigine) / scala;
    return valoare;
}

// scalarea pe verticala
private int scalaY(double y) {
    int pixelCoord;
    pixelCoord = (int) Math.round(-y * scala) + yOrigine;
    return pixelCoord;
}

// modificarea parametrilor a si b
public void seteazaParametri(int aVal, int bVal) {
    a = (aMax + aMin)/2 + (aVal - 50) * (aMax - aMin) / 100;
    b = (bMax + bMin)/2 + (bVal - 50) * (bMax - bMin) / 100;
}

// desenarea axei Ox
private void abscisa(Graphics g) {
    g.drawLine(xPixelStart, yOrigine, xPixelEnd, yOrigine);
    g.drawLine(xPixelEnd - 5, yOrigine - 5, xPixelEnd,
        yOrigine);
    g.drawLine(xPixelEnd - 5, yOrigine + 5, xPixelEnd,
        yOrigine);
    int i, iScala;
    for (i = 1; i * (int) scala < xOrigine - xPixelStart; i++) {
        iScala = i * (int) scala;
        g.drawLine(xOrigine + iScala, yOrigine,
            xOrigine + iScala, yOrigine - 2);
        g.drawLine(xOrigine - iScala, yOrigine,
            xOrigine - iScala, yOrigine - 2);
    }
}

// desenarea axei Oy
private void ordonata(Graphics g) {
    g.drawLine(xOrigine, yPixelEnd, xOrigine, yPixelStart);
    g.drawLine(xOrigine - 5, yPixelStart + 5, xOrigine,
        yPixelStart);
    g.drawLine(xOrigine + 5, yPixelStart + 5, xOrigine,
        yPixelStart);
    int i, iScala;

```

```

for (i = 1; i * (int) scala < yOrigine - yPixelStart; i++) {
    iScala = i * (int) scala;
    g.drawLine(xOrigine, yOrigine - iScala,
        xOrigine - 2, yOrigine - iScala);
    g.drawLine(xOrigine, yOrigine + iScala,
        xOrigine - 2, yOrigine + iScala);
}
}

// desenarea graficului propriu-zis
public void deseneaza(Graphics g) {
    double x, y, nextX, nextY;
    int xPixel, yPixel, nextXPixel, nextYPixel;
    g.drawString(" a="+a+" b="+b, 30, 60);
    abscisa(g);
    ordonata(g);
    for (xPixel = xPixelStart; xPixel < xPixelEnd; xPixel++) {
        x = scalaX(xPixel);
        y = functie(x, a, b);
        yPixel = scalaY(y);
        nextXPixel = xPixel + 1;
        nextX = scalaX(nextXPixel);
        nextY = functie(nextX, a, b);
        nextYPixel = scalaY(nextY);
        g.drawLine(xPixel, yPixel, nextXPixel, nextYPixel);
    }
}

```

Se observă din metoda deseneaza() că graficul funcției este construit din linii foarte scurte.

7.4.7. Butoane de selecție (check-box și radio)

Clasa care corespunde unui câmp de tip check-box este `java.awt.Checkbox`. Aceasta este o componentă care poate fi selectată sau nu, true sau false. Cei mai utilizați constructori sunt:

- `public Checkbox(String eticheta) throws HeadlessException`
- `public Checkbox(String eticheta, boolean stare) throws HeadlessException`

Ambii creează un check-box cu eticheta specificată. Prima variantă stabilește starea inițială ca fiind neselectată, iar cealaltă variantă va crea un check-box selectat inițial, dacă stare va fi true, sau neselectat, în caz contrar.

Pentru a capta acțiunile asupra câmpului check-box va trebui ca acesta să aibă atașat un ascultător de tip `ItemListener`. Acest lucru (atașarea) se realizează prin apelul metodei `addItemListener()` din clasa `Checkbox`.

Interfața `ItemListener` are o singură metodă:

- `public void itemStateChanged(ItemEvent e)`

și este apelată ori de câte ori starea check-box-ului este modificată.

Clasa `Checkbox` pune la dispoziție posibilitatea modificării/obținerii etichetei și respectiv a stării prin metodele:

- `public String getLabel()`
- `public boolean getState()`
- `public void setLabel(String etichetaNoua)`
- `public void setState(boolean stareNoua)`

Java pune la dispoziție gruparea mai multor check-box-uri. Pentru aceasta există clasa `CheckboxGroup`. Aceasta permite ca la un moment dat un singur câmp check-box să fie selectat, iar celelalte nu. Selectarea unuia dintre ele duce la deselectarea celui care a fost selectat anterior. Singurele lucruri care diferă față de check-box-uri sunt definirea (crearea) și rezolvarea evenimentelor.

Pentru a crea un grup de check-box-uri se utilizează constructorul implicit al clasei `CheckboxGroup`. Adăugarea unui check-box se realizează fie prin apelarea constructorului

```
public Checkbox(String eticheta, CheckboxGroup grup, boolean state),
```

fie a metodei

```
public setCheckboxGroup(CheckboxGroup grup)
```

Obținerea grupului din care face parte un check-box se realizează în urma apelului metodei `getCheckboxGroup()`.

Exemplul 7.4.10. Crearea unui grup de check-box-uri.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ButonRadio extends Applet
    implements ItemListener {
    // declararea grupului de checkbox-uri
    private CheckboxGroup grup;
    // declararea checkbox-urilor
    private Checkbox red, orange, yellow;

    public void init() {
        // creare unui grup de checkbox-uri
```

```
grup = new CheckboxGroup();

// crearea unui checkbox si adaugarea la grup
red = new Checkbox("Red", grup, false);
// adaugarea la applet
add(red);
// stabilirea ascultatorului
red.addItemListener(this);

orange = new Checkbox("Orange", grup, true);
add(orange);
orange.addItemListener(this);

yellow = new Checkbox("Yellow", grup, false);
add(yellow);
yellow.addItemListener(this);
}

// metoda de interceptare a evenimentelor
// legate de checkbox-uri
public void itemStateChanged(ItemEvent e) {
    // daca s-a actionat asupra unui checkbox
    // care prezinta interes
    if ((e.getSource() == red) || (e.getSource() == orange)
        || (e.getSource() == yellow))
        // se va afisa un mesaj in bara de stare a navigatorului
        showStatus("Starea = " + red.getState() + " " +
                    orange.getState() + " " + yellow.getState());
}
```

Se observă în bara de stare a navigatorului că la un moment dat un singur check-box este selectat.

7.4.8. Liste de opțiuni

O listă de opțiuni este o listă de elemente text din care utilizatorul le poate alege pe cele dorite. Există două tipuri de liste de opțiuni după numărul de elemente care pot fi selectate la un moment dat: simple (se poate alege o singură opțiune) și complexe (se pot alege mai multe opțiuni).

O listă de opțiuni simplă este asemănătoare cu un grup de check-box-uri, în sensul că la un moment dat o singură opțiune este selectată, iar selectarea unei opțiuni duce la deselectarea celei selectate anterior. Deosebirea constă în faptul că o singură opțiune este vizibilă (cea selectată), iar în momentul în care utilizatorul dorește selectarea altei opțiuni, se vor afișa și celelalte.

O listă de opțiuni simplă se realizează prin instanțierea clasei `Choice`. Aceasta are doar un constructor implicit. Ascultătorii acestei componente grafice trebuie să implementeze interfața `ItemListener`.

Cele mai importante metode ale clasei `Choice` sunt explicate în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
<code>public void add(String optiune)</code>	Adaugă o opțiune dată de opțiune la listă.
<code>public void addItemListener (ItemListener ascultator)</code>	Stabilește un ascultător pentru lista de opțiuni.
<code>public String getItem(int pozitie)</code>	Returnează opțiunea de pe poziția specificată.
<code>public int getItemCount()</code>	Întoarce numărul de opțiuni din listă.
<code>public int getSelectedIndex()</code>	Returnează indexul opțiunii selectate.
<code>public String getSelectedItem()</code>	Returnează opțiunea selectată.
<code>public void insert(String optiune, int pozitie)</code>	Inserează opțiunea specificată pe poziția indicată. Dacă există un element pe poziția respectivă, atunci opțiunile următoare sunt deplasate cu o poziție, altfel elementul va fi inserat pe prima poziție. Dacă elementului selectat i s-a schimbat indexul, atunci opțiunea nouă inserată va deveni cea selectată.
<code>public void remove(int pozitie)</code>	Se elimină opțiunea de pe poziția specificată, iar cele ce urmează vor fi deplasate cu o poziție. Dacă s-a eliminat opțiunea selectată, atunci prima opțiune va deveni selectată.
<code>public void remove(String optiune)</code>	Analog cu metoda precedentă cu deosebirea că se va elimina opțiunea specificată.
<code>public void removeAll()</code>	Se vor elimina toate opțiunile din listă.
<code>public void select(int pozitie)</code>	Se selectează opțiunea specificată prin poziția sa.
<code>public void select(String optiune)</code>	Se selectează opțiunea indicată.

Exemplul 7.4.11. Appletul va crea o listă de opțiuni, iar în bara de stare a navigatorului va fi afișată opțiunea selectată.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ListaOptiuni extends Applet
    implements ItemListener {
    private Choice colourChoice;
```

```
public void init() {
    // crearea unei liste de optiuni
    colourChoice = new Choice();
    // adaugarea optiunilor
    colourChoice.add("Rosu");
    colourChoice.add("Galben");
    colourChoice.add("Albastru");
    // adaugarea listei de optiuni la applet
    add(colourChoice);
    // stabilirea ascultatorului
    colourChoice.addItemListener(this);
}

// metoda de tratare a evenimentelor legate
// de lista de optiuni
public void itemStateChanged(ItemEvent e) {
    // daca lista de optiuni este sursa evenimentului
    if (e.getSource() == colourChoice) {
        String aChoice = e.getItem().toString();
        // se afiseaza opțiunea selectata in status bar
        showStatus("Starea = " + aChoice);
    }
}
```

O listă de opțiuni complexă este o listă de opțiuni din care unul sau mai multe elemente pot fi selectate. Clasa `List` pune la dispoziție crearea unei liste de opțiuni care se poate comporta ca una simplă sau ca una complexă. Implicit este simplă.

Pe lângă cel implicit, clasa `List` mai posedă doi constructori frecvent utilizați:

- `public List(int randuri) throws HeadlessException`
- `public List(int randuri, boolean tip) throws HeadlessException`

Parametrul `randuri` se referă la numărul de opțiuni care vor fi vizibile la un moment dat, iar dacă `tip` are valoarea `true`, atunci lista de opțiuni va fi complexă. Implicit este simplă.

Cele mai importante metode ale clasei `List` sunt următoarele:

Prototipul metodei	Descrierea metodei
<code>public void add(String optiune)</code>	Adaugă opțiunea indicată la sfârșitul listei.
<code>public void add(String optiune, int pozitie)</code>	Adaugă opțiunea indicată pe poziția specificată. Dacă indexul nu este valid, atunci opțiunea se adaugă la sfârșitul listei.
<code>public void addActionListener (ActionListener ascultator)</code>	Adaugă un ascultător de tip <code>ActionListener</code> pentru a trata evenimentele apărute în urma unui dublu click.

Prototipul metodei	Descrierea metodei
<code>public void addItemListener (ItemListener ascultator)</code>	Adaugă un ascultător de tip <code>ItemListener</code> pentru a trata evenimentele apărute în urma unui click.
<code>public void deselect(int index)</code>	Deselectează opțiunea cu indexul specificat.
<code>public String getItem(int index)</code>	Returnează opțiunea care are indexul specificat.
<code>public int getItemCount()</code>	Returnează numărul de opțiuni din lista de opțiuni.
<code>public String[] getItems()</code>	Întoarce un tablou de tip <code>String</code> cu toate opțiunile din listă.
<code>public int getRows()</code>	Returnează numărul de linii vizibile.
<code>public int[] getSelectedIndexes()</code>	Întoarce un tablou cu indecșii opțiunilor selectate.
<code>public String[] getSelectedItems()</code>	Întoarce un tablou cu opțiunile selectate.
<code>public void isIndexSelected(int index)</code>	Testează dacă opțiunea cu indexul indicat este selectată.
<code>public void isMultipleMode()</code>	Întoarce <code>true</code> dacă se permite selectarea multiplă și <code>false</code> în caz contrar.
<code>public void makeVisible(int index)</code>	Face vizibilă opțiunea cu indexul specificat.
<code>public void remove(int index)</code>	Elimină opțiunea cu indexul specificat.
<code>public void remove(String optiune)</code>	Elimină opțiunea indicată.
<code>public void removeAll()</code>	Elimină toate opțiunile din listă.
<code>public void select(int index)</code>	Selectează opțiunea cu indexul specificat.
<code>public void setMultipleMode (boolean b)</code>	Dacă <code>b</code> are valoarea <code>true</code> , atunci lista va deveni complexă, iar în caz contrar simplă.

Exemplul 7.4.11. Următorul applet conține o listă cu 4 opțiuni cu selectare multiplă. La un moment dat doar 3 opțiuni sunt vizibile (va apărea și o bară de defilare pentru vizualizarea celeilalte opțiuni). La apăsarea butonului Gata se vor afișa în bara de stare a navigatorului elementele selectate.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ListaOptiuniComplexa
    extends Applet implements ActionListener {
    private List list;
```

```
private Button gata;

public void init() {
    // crearea unei liste de optiuni
    // cu selectare multipla
    list = new List(3, true);
    // adaugarea optiunilor
    list.add("Red");
    list.add("Yellow");
    list.add("Blue");
    list.add("Green");
    // adaugarea listei la applet
    add(list);

    // crearea unui buton
    gata = new Button("Gata");
    add(gata);
    gata.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == gata) {
        // obtinerea elementelor selectate
        String[] selectie = list.getSelectedItems();
        String sir="";
        for (int i = 0; i < selectie.length; i++)
            sir = sir + selectie[i] + " ";
        // afisarea elementelor selectate
        // in bara de stare
        showStatus("Starea = " + sir);
    }
}
```

7.4.9. Suprafețe de desenare (cavasses)

Până acum am văzut aplicații în care desenul și butoanele erau în aceeași fereastră. Problema este că butoanele ar putea acoperi desenul. Suprafețele de desenare (pânzele, eng. *cavasses*) rezolvă această problemă definind o arie separată unde se plasează desenul. O suprafață de desenare este creată ca un obiect al unei clase derivate din clasa `java.awt.Canvas`. La rândul ei, clasa `Canvas` este derivată din clasa `Component`. Programatorul poate suprascrie funcția `paint()`.

Exemplul 7.4.12. Adăugarea unei suprafețe de desenare la un applet. Aceasta va avea culoarea gri și un mesaj de probă.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluCanvas extends Applet {
    // crearea unui canvas propriu
    private CanvasulMeu unCanvas = new CanvasulMeu();

    public void init() {
        // stabilirea culorii de fundal pentru canvas
        unCanvas.setBackground(Color.gray);
        // redimensionarea canvas-ului
        unCanvas.setSize(300, 200);
        // adaugarea canvas-ului la applet
        add(unCanvas);
    }

    // definirea propriului canvas
    class CanvasulMeu extends Canvas {
        // redefinirea functiei de desenare
        public void paint(Graphics g) {
            g.drawString("Proba de canvas!", 15, 15);
        }
    }
}
```

Bineînțeles că se pot adăuga mai multe suprafețe de desenare la un applet.

7.4.10. Panouri (panels)

Panourile sunt utilizate pentru gruparea unui număr de componente, spre deosebire de suprafețele grafice care sunt folosite pentru afișarea textului sau graficii. Dacă dorim un grup de butoane plasate în partea de sus a ferestrei și alt grup în partea de jos a ferestrei, atunci putem crea două panouri pentru fiecare dintre aceste grupuri. Unui panou îi corespunde un obiect din clasa `java.awt.Panel`. Această clasă este derivată din `Container`.

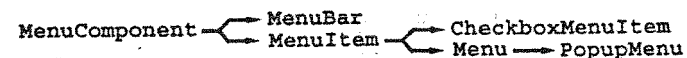
Exemplul 7.4.13. Secvența de cod ilustrează crearea unui panou, adăugarea de componente la panou și adăugarea acestuia la un applet.

```
// crearea unui panou
Panel p = new Panel();
// adaugarea componentelor la panou
Button b1 = new Button("Apasa 1");
```

```
Button b2 = new Button("Apasa 2");
p.add(b1);
p.add(b2);
// adaugarea panoului la fereastra appletului
add(p);
```

7.4.11. Meniuri Popup

În acest capitol vom prezenta doar meniurile pop-up, întrucât doar acestea pot fi utilizate de către appleturi, iar în aplicațiile de sine stătătoare se utilizează meniurile din biblioteca Swing prezentată în capitolul 14.



În figura de mai sus este reprezentată ierarhia claselor utilizate pentru construirea meniurilor în AWT.

Unui meniu popup îi corespunde clasa `PopupMenu`. Aceasta are doi constructori: cel implicit și unul cu un parametru de tip `String` care stabilește un nume pentru meniu. Cea mai importantă metodă este:

- `public void show(Component origine, int x, int y)`

unde `x` și `y` constituie coordonata la care se va afișa meniul (colțul din stânga-sus), iar `origine` este componenta pentru care este definit spațiul de coordonate. Această metodă duce la vizualizarea meniului.

Clasa `PopupMenu` extinde clasa `Menu`. Aceasta posedă următoarele metode importante:

Prototipul metodei	Descrierea metodei
<code>public MenuItem add(MenuItem menuItem)</code>	Adaugă elementul meniului specificat. Returnează elementul adăugat.
<code>public void add(String label)</code>	Adaugă un element de meniu cu textul specificat.
<code>public void insert(MenuItem menuItem, int index)</code>	Inserează elementul meniului pe poziția indicată.
<code>public void insert(String label, int index)</code>	Inserează un element de meniu cu textul specificat pe poziția indicată.
<code>public MenuItem getItem(int index)</code>	Returnează elementul care are indexul specificat.
<code>public int getItemCount()</code>	Întoarce numărul de elemente din meniu.
<code>public void addSeparator()</code>	Adaugă un separator.
<code>public void insertSeparator(int index)</code>	Inserează un separator pe poziția specificată.
<code>public void remove(int index)</code>	Elimină elementul meniului de pe poziția indicată.
<code>public void remove(MenuItem item)</code>	Elimină elementul meniului specificat.
<code>public void removeAll()</code>	Elimină toate elementele din meniu.

Clasa Menu este derivată din MenuItem. Aceasta este clasa corespunzătoare unui element dintr-un meniu. Clasa MenuItem posedă următoarele metode importante:

Prototipul metodei	Descrierea metodei
public String getLabel ()	Obține eticheta elementului meniului.
public void setLabel (String label)	Stabilește eticheta pentru element.
public boolean isEnabled ()	Testează dacă elementul din meniu este disponibil.
public void setEnabled (boolean b)	Stabilește dacă elementul din meniu este disponibil.

Pentru un element al meniului se poate atașa un ascultător de tip ActionListener cu apelul metodei binecunoscute addActionListener().

Există un tip particular de element al meniului: de tip checkbox. Acesta are două stări: selectat (true) sau neselectat (false) și este reprezentat de clasa CheckboxMenuItem care are următoarele metode semnificative:

Prototipul metodei	Descrierea metodei
public void addItemListener (ItemListener l)	Stabilește un ascultător pentru elementul de tip checkbox al meniului.
public boolean getState ()	Obține starea elementului.
public void setState (boolean b)	Setează starea elementului.

Interfața ItemListener conține o singură metodă:

- public void itemStateChanged(ItemEvent e)

care este invocată când un element este selectat sau deselectat.

Exemplul 7.4.14. Appletul de mai jos evidențiază modalitatea de creare a meniurilor popup precum și utilizarea celor două tipuri de elemente ale meniurilor:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluMenuPopup extends Applet
    implements ActionListener, ItemListener {
    private PopupMenu meniu;
    private Button b;
    private boolean state;
    private String mesaj =
        "Acesta este un exemplu de meniu popup.";

    private void addOption(String opt) {
        MenuItem item = new MenuItem(opt);
        item.addActionListener(this);
```

```
        meniu.add(item);
    }

    public void init() {
        // crearea meniului
        meniu = new PopupMenu("Meniu");
        add(meniu);
        // crearea optiunilor
        addOption("Alb");
        addOption("Verde");
        addOption("Gri deschis");
        meniu.addSeparator();
        CheckboxMenuItem check = new CheckboxMenuItem("Mesaj");
        check.addItemListener(this);
        meniu.add(check);

        // crearea butonului
        b = new Button("Afisare meniu");
        b.addActionListener(this);
        add(b);

        // initializari
        state = false;
        setForeground(Color.white);
    }

    public void paint(Graphics g) {
        g.drawString(mesaj, 100, 150);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == b)
            // afisarea meniului
            meniu.show(this, 10, 10);
        else {
            // modificarea culorii fundalului
            MenuItem item = (MenuItem) e.getSource();
            String label = item.getLabel();
            if (label.equals("Verde"))
                setBackground(Color.green);
            else if (label.equals("Alb"))
                setBackground(Color.white);
            else
                setBackground(Color.lightGray);
            if (!state)
```

```

        setBackground(getBackground());
    }

    public void itemStateChanged(ItemEvent e) {
        // schimbarea culorii de scriere
        if (state)
            setBackground(getBackground());
        else
            setBackground(Color.black);
        // modificarea starii
        state = ! state;
    }
}

```

7.5. Gestionări de poziționare (Layout Managers)

Un gestionar de poziționare este un obiect al unei clase care implementează interfața `LayoutManager`.

Pentru stabilirea unui manager de proiecte, clasa `Container` pune la dispoziție metoda

```
• public void setLayout(LayoutManager m)
```

unde `LayoutManager` este o interfață care este implementată de toți managerii de proiecte.

Orice container are un gestionar de poziționare. Java pune la dispoziție următoarele clase pentru gestionarea poziționării:

- `BorderLayout`
- `CardLayout`
- `FlowLayout`
- `GridBagLayout`
- `GridLayout`

7.5.1. Gestionarul `BorderLayout`

Acest gestionar de poziționare conține cinci regiuni: nord, est, sud, vest și centru. Fiecare regiune poate conține cel mult o componentă (adăugarea unei componente implică eliminarea celei precedente). Pentru identificarea unei regiuni se utilizează atributele publice și statice ale clasei `BorderLayout` (respectiv pentru nord, est, sud, vest și centru):

- `public static String NORTH`
- `public static String EAST`
- `public static String SOUTH`

- `public static String WEST`
- `public static String CENTER`

Exemplul 7.5.1. Adăugarea unui buton în partea de jos a unui panou se poate realiza astfel:

```

// crearea panoului
Panel panou = new Panel();
// stabilirea gestionarului de pozitionare
panou.setLayout(new BorderLayout());
// crearea unui buton
Button butonOk = new Button("OK");
butonOk.addActionListener(this);
// pozitionarea butonului in partea de jos a panoului
add(butonOk, BorderLayout.SOUTH);

```

Dacă nu se specifică zona în care se va adăuga o componentă, atunci se va poziționa în centru. Considerând că panou este obiect al clasei `Panel` și are un gestionar de poziționare `BorderLayout`, linia de cod

```
panou.add(componenta);
```

este echivalentă cu

```
panou.add(componenta, BorderLayout.CENTER);
```

O altă variantă de a adăuga o componentă într-o anumită zonă este de a specifica mai întâi regiunea sub forma unui șir de caractere și apoi componenta.

Exemplul 7.5.2. Adăugarea unor componente în cele cinci regiuni (respectiv nord, sud, vest, est și centru) ale gestionarului `BorderLayout`.

```

add("North", componenta1);
add("South", componenta2);
add("West", componenta3);
add("East", componenta4);
add("Center", componenta5);

```

Exemplul 7.5.3. Appletul va poziționa câte un buton în regiunile est, vest, centru și sud, iar la acționarea butoanelor va apărea un mesaj în regiunea nordică.

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluGestionarBorderLayout
    extends Applet implements ActionListener {
    Panel bottom, centrat;

```

```

Button sud, vest, est, centru;
CanvasNordic nord;

public void init() {
    // stabilirea gestionarului de pozitionare
    setLayout(new BorderLayout());

    // crearea butoanelor
    sud = new Button("Sud");
    vest = new Button("Vest");
    est = new Button("Est");
    centru = new Button("Centru");

    // atasarea ascultatorului
    sud.addActionListener(this);
    vest.addActionListener(this);
    est.addActionListener(this);
    centru.addActionListener(this);

    // adaugarea butoanelor
    add("West", vest);
    add("East", est);

    // adaugarea de panouri
    bottom = new Panel();
    bottom.add(sud);
    add("South", bottom);
    centrat = new Panel();
    centrat.add(centru);
    add(centrat);

    // adaugarea canvasului
    nord = new CanvasNordic();
    nord.setBackground(Color.black);
    nord.setForeground(Color.white);
    nord.setSize(200, 40);
    add(nord, BorderLayout.NORTH);
}

// afisarea in partea nordica a unui mesaj
public void actionPerformed(ActionEvent e) {
    Button buton = (Button) e.getSource();
    nord.mesaj = "Ultima data ati apasat pe " +
        buton.getLabel();
    nord.repaint();
}

```

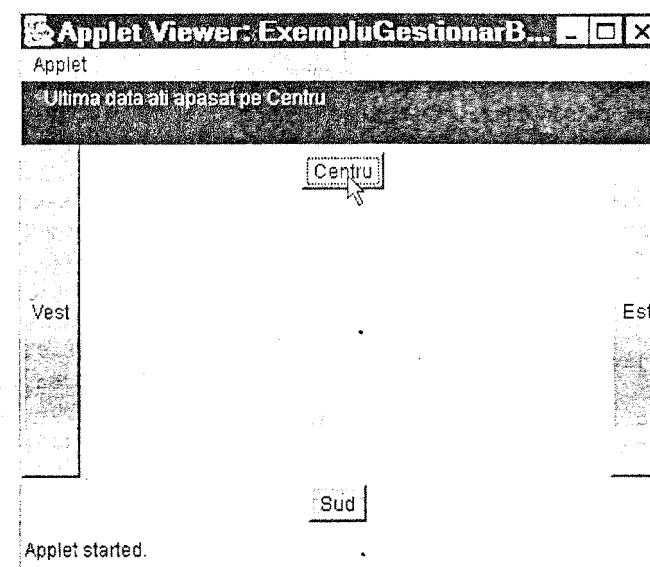
```

}
}

// definirea propriului canvas
class CanvasNordic extends Canvas {
    String mesaj="";
    // redefinirea functiei de desenare
    public void paint(Graphics g) {
        g.drawString(mesaj, 15, 15);
    }
}

```

În urma execuției appletului putem avea situația din figura de mai jos (când s-a apăsât pe butonul Centru):



Butoanele Est și Vest au fost adăugate direct în regiuni și de aceea au ocupat întreaga regiune. Butoanele Centru și Sud au fost incluse prin intermediul panourilor (panourile au fost incluse în regiuni, iar butoanele în panouri) și din acest motiv apar cu dimensiunile standard.

7.5.2. Gestionarul CardLayout

Acest gestionar permite afișarea unei singure componente la un moment dat. De obicei, se adaugă panouri care conțin componentele dorite, iar pentru fiecare panou putem să stabilim gestionarul de poziționare dorit.

Clasa `CardLayout` posedă doi constructori: unul implicit și unul cu doi parametri de tip `int` care indică marginea care este lăsată la stânga și la dreapta, respectiv sus și jos:

- `public CardLayout(int orizontal, int vertical)`

Cele mai semnificative metode din clasa `CardLayout` sunt:

Prototipul metodei	Descrierea metodei
<code>void addLayoutComponent(String nume, Component componenta)</code>	Adaugă componenta specificată și stabilește pentru aceasta un nume (dat de parametrul <code>nume</code>).
<code>void first(Container parinte)</code>	Se afișează prima componentă.
<code>void last(Container parinte)</code>	Se afișează ultima componentă.
<code>void next(Container parinte)</code>	Se afișează componenta următoare.
<code>void previous(Container parinte)</code>	Se afișează componenta precedentă.
<code>void show(Container parinte, String nume)</code>	Se activează componenta care are atașat numele specificat.

Menționăm că parametrul `parinte` este referința la obiectul care are stabilit gestionarul `CardLayout`.

Exemplul 7.5.4. Appletul are un gestionar de poziționare de tip `CardLayout` și conține trei componente (panouri mai exact). La apăsarea butoanelor se trece de la o componentă la alta.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluGestionarCardLayout
    extends Applet implements ActionListener {

    Button next1, next2, next3;
    Panel p1, p2, p3;
    TextArea t1, t2, t3;
    CardLayout gestionar;
    int pagina = 1;

    public void init() {
        gestionar = new CardLayout();
        setLayout(gestionar);

        // crearea butoanelor
        next1 = new Button("Mai departe...");
        next2 = new Button("Mai departe...");
        next3 = new Button("La inceput");
```

```
        next1.addActionListener(this);
        next2.addActionListener(this);
        next3.addActionListener(this);

        // crearea panourilor
        p1 = new Panel();
        p2 = new Panel();
        p3 = new Panel();

        // adaugarea panourilor
        add("P1", p1);
        add("P2", p2);
        add("P3", p3);

        // completarea primului panou
        t1 = new TextArea("Aceasta este prima pagina.\n" +
            "Apasati butonul din dreapta.",
            10, 40, TextArea.SCROLLBARS_NONE);
        t1.setEditable(false);
        p1.add(t1);
        p1.add(next1);

        // completarea celui de-al doilea panou
        t2 = new TextArea("Ati ajuns la pagina a doua.\n" +
            "Pentru a merge mai departe apasati butonul alaturat",
            10, 30, TextArea.SCROLLBARS_NONE);
        t2.setEditable(false);
        p2.add(t2);
        p2.add(next2);

        // completarea celui de-al treilea panou
        p3.add(next3);
    }

    public void actionPerformed(ActionEvent e) {
        gestionar.next(this);
    }
}
```

7.5.3. Gestionarul `FlowLayout`

Gestionarul de poziționare `FlowLayout` este implicit pentru containerele `Panel` și `Applet`. Acesta conține o singură regiune în care componentele sunt aliniate centrat și sunt plasate de la stânga la dreapta. În cazul în care sunt prea mult componente pe

un singur rând, atunci fie se redimensionează containerul părinte, fie se trece la rândul următor.

La rularea programelor din exemplele 7.5.3. și 7.5.4. se poate observa modul de adăugare a componentelor într-un container care are un gestionar `FlowLayout`.

Implicit, componentele sunt aliniate centrat. Metoda `setAlignment()` stabilește alinierea rândurilor cu componentele. Aceasta are un singur parametru întreg și poate lua următoarele valori predefinite: `FlowLayout.LEFT` (pentru aliniere la stânga), `FlowLayout.RIGHT` (pentru aliniere la dreapta) și `FlowLayout.CENTER` (pentru aliniere centrată). Componentele se vor adăuga de la stânga la dreapta, indiferent de tipul alinierii.

Exemplul 7.5.5. Appletul va afișa o etichetă aliniată la dreapta.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ExempluGestionarFlowLayout
    extends Applet {

    private Label l;

    public void init() {
        // crearea si adaugarea unei etichete
        l = new Label("Aliniere la dreapta.");
        add(l);
        // obtinerea gestionarului
        FlowLayout gestionar = (FlowLayout) getLayout();
        // alinierea la dreapta
        gestionar.setAlignment(FlowLayout.RIGHT);
    }
}
```

7.5.4. Gestionarul GridLayout

Gestionarul de poziționare `GridLayout` conține o grilă de celule, toate cu aceleași dimensiuni. Fiecare componentă este adăugată într-o celulă de la stânga la dreapta și de sus în jos. Această ordine poate fi schimbată prin apelul metodei `setComponentOrientation()` din clasa `Component` (mai exact, al obiectului care are stabilit acest gestionar de poziționare).

Constructorii clasei `GridLayout` (în afară de cel implicit) sunt:

- `public GridLayout(int nrRanduri, int nrColoane)`
- `public GridLayout(int nrRanduri, int nrColoane, int orizontal, int vertical)`

unde primii doi parametri se referă la numărul de rânduri și coloane al grilei, iar `oriz` și `vert`, la distanța dintre celule pe orizontală și respectiv pe verticală.

Dacă numărul de componente depășește dimensiunea grilei, aceasta se va redimensiona corespunzător, pentru a putea încăpea toate elementele sale.

Exemplul 7.5.6. Appletul afișează rezolvarea unui rebus de 4 linii și 4 coloane.

```
import java.applet.Applet;
import java.awt.*;

public class ExempluGestionarGridLayout extends Applet {
    public Label rebus[][] = new Label[4][4];

    public void init() {
        // stabilirea gestionarului GridLayout
        setLayout(new GridLayout(4,4,3,3));
        // crearea etichetelor
        int i,j;
        String[] litere = {
            "J", "A", "V", "A",
            "A", "R", " ", "R",
            "V", " ", "A", "C",
            "A", "R", "I", "E" };
        for(i=0;i<4;i++)
            for(j=0;j<4;j++) {
                rebus[i][j] = new Label();
                rebus[i][j].setText(" "+litere[4*i+j]);
                rebus[i][j].setBackground(Color.lightGray);
                add(rebus[i][j]);
            }
        // colorarea patratelor neutilizate
        rebus[2][1].setBackground(Color.black);
        rebus[1][2].setBackground(Color.black);
        // stabilirea dimensiunii appletului
        setSize(100,130);
    }
}
```

La execuția appletului va apărea:

J	A	V	A
A	R		R
V		A	C
A	R	I	E

7.5.5. Gestionarul GridBagLayout

Gestionarul GridBagLayout este unul mai flexibil și permite alinierea verticală și orizontală a componentelor pe baza unor constrângeri. Gestionarul este asemănător cu GridLayout cu deosebirea că o componentă se poate „întinde” pe mai multe celule. Clasa care permite specificarea constrângerilor este GridBagConstraints. Setarea și respectiv obținerea constrângerilor pentru o componentă se realizează cu ajutorul următoarelor metode din clasa GridBagLayout:

- public void setConstraints(Component comp, GridBagConstraints c);
 - public GridBagConstraints getConstraints(Component comp);
- unde comp este componenta în cauză, iar c specifică constrângerile ce vor fi aplicate componentei.

Cele mai importante atribute din clasa GridBagConstraints, utile pentru stabilirea poziției componentei în cadrul grilei, sunt prezentate în tabelul de mai jos.

Atribut	Descrierea atributului
int gridheight	Indică numărul de celule pe verticală pe care îl ocupă componenta.
int gridwidth	Indică numărul de celule pe orizontală pe care îl ocupă componenta.
int gridx	Indică rândul pe care va fi situată componenta.
int gridy	Indică coloana pe care va fi situată componenta.

Următorul exemplu prezintă modalitatea de a utiliza gestionarul GridBagLayout, precum și stabilirea constrângerilor pentru o cât mai bună aliniere a componentelor.

Exemplul 7.5.7. Appletul utilizează gestionarul GridBagLayout.

```
import java.applet.Applet;
import java.awt.*;

public class ExempluGestionarGridBagLayout extends Applet {
    public void init() {
        // stabilirea gestionarului de pozitionare
        GridBagLayout grid = new GridBagLayout();
        setLayout(grid);

        // crearea componentelor
        Label l1 = new Label("Nume: ");
        Label l2 = new Label("Prenume: ");
        TextField t1 = new TextField(25);
        TextField t2 = new TextField(25);
        Checkbox cb = new Checkbox("Tanar", true);

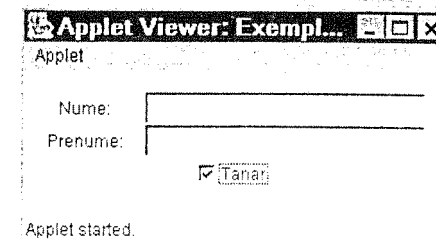
        // stabilirea constrangerilor
        GridBagConstraints c = new GridBagConstraints();
        // se stabileste prima coloana
```

```
c.gridx = 0;
// se stabileste primul rand
c.gridy = 0;
// componenta va ocupa 2 celule pe orizontala
c.gridwidth = 2;
// stabilirea constrangerilor pentru l1
grid.setConstraints(l1,c);
// se stabileste al doilea rand
c.gridy = 1;
// stabilirea constrangerilor pentru l2
grid.setConstraints(l2,c);
// se va stabili a treia coloana
c.gridx = 2;
grid.setConstraints(t1,c);
// se stabileste primul rand
c.gridy = 0;
grid.setConstraints(t2,c);
// se va stabili coloana a doua, randul 3
c.gridx = 1;
c.gridy = 2;
// se va ocupa 3 celule pe orizontala
c.gridwidth = 3;
grid.setConstraints(cb,c);

// adaugarea componentelor
add(cb);
add(l1);
add(l2);
add(t1);
add(t2);

// stabilirea dimensiunii appletului
setSize(300,100);
}
```

Appletul va afișa următoarele componente (care nu au efect, întrucât nu le-au fost atașați ascultători):



7.5.6. Poziționarea absolută

Pentru a plasa componentele fără a utiliza un gestionar predefinit, vom apela metoda `setLayout()` cu valoarea `null`:

```
setLayout(null);
```

Pentru dimensionarea și plasarea componentelor putem utiliza metoda:

- `void setBounds(int x, int y, int latime, int inaltime);`

unde `x` și `y` reprezintă coordonatele poziției componentei în container, iar parametrii `latime` și `inaltime` reprezintă dimensiunile componentei. Același lucru se poate realiza și cu metodele `setLocation()` și `setSize()`:

- `void setLocation(int x, int y);`
- `void setSize(int latime, int inaltime);`

unde parametrii au aceeași semnificație ca mai sus.

Exemplul 7.5.8. Appletul afișează trei butoane așezate în diagonală.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class PozitionareAbsoluta extends Applet
    implements ActionListener {

    String sir = "";
    String butonSelectat;
    Button b1, b2, b3;

    public void init() {
        // stabilirea pozitionarii absolute
        setLayout(null);

        // crearea de butoane
        b1 = new Button("Buton1");
        b2 = new Button("Buton2");
        b3 = new Button("Buton3");
        // adaugarea la applet
        add(b1);
        add(b2);
        add(b3);
        // stabilirea dimensiunilor si a pozitiei
        b1.setBounds( 0, 0, 60, 30);
        b2.setBounds( 60, 30, 60, 30);
        b3.setBounds(120, 60, 60, 30);
```

```
// stabilirea ascultatorilor
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
}

public void actionPerformed(ActionEvent ev) {
    Button b = (Button) ev.getSource();
    sir = "Ati apasat " + b.getLabel();
    repaint();
}

public void paint(Graphics g) {
    g.drawString(sir, 10, 120);
}
}
```

7.6. Ferestre

Ferestrele în AWT au la bază clasa `java.awt.Window`. Aceasta este derivată din `Container` și nu conține implicit margini sau vreo bară de meniuri sau unelte. Această clasă nu prea este utilizată direct pentru construirea ferestrelor. Sunt utilizate clasele derivate, cum ar fi `Dialog` (pentru ferestre de dialog) sau `Frame` (pentru fereastra unei aplicații).

7.6.1. Clasa `java.awt.Window`

De remarcat faptul că ferestrele formează o ierarhie. Fiecare fereastră are o fereastră părinte și posedă (sau nu) mai multe subferestre.

Cele mai importante metode publice din clasa `Window` sunt următoarele:

Protonul metodei	Descrierea metodei
<code>void addWindowListener(WindowListener ascultator)</code>	Stabilește un ascultător pentru evenimentele legate de fereastră.
<code>void dispose()</code>	Eliberează resursele native ale ferestrei și implică dispariția acesteia de pe ecran.
<code>Component getFocusOwner()</code>	Returnează componenta activă din fereastră sau <code>null</code> , dacă nici o componentă nu este activă.
<code>Window[] getOwnedWindows()</code>	Returnează un vector cu subferestrele ferestrei curente.
<code>Window getOwner()</code>	Întoarce părintele acestei ferestre.
<code>void hide()</code>	Ascunde fereastra cu toate componentele sale.

Prototipul metodei	Descrierea metodei
boolean isActive()	Testează dacă fereastra este activă (pentru Frame sau Dialog).
boolean isFocused()	Testează dacă fereastra este activă.
boolean isShowing()	Testează dacă fereastra este vizibilă pe ecran.
void pack()	Fereastra și componentele sale vor avea dimensiunile optime.
void show()	Afișează fereastra pe ecran.
void toBack()	Trimite fereastra în fundal și aceasta devine inactivă.
void toFront()	Aduce fereastra în prim-plan și aceasta devine activă.

Aceste metode se vor regăsi și la clasele derivate cu aceleași funcționalități.

Interfața `WindowListener` definește șapte metode. Acestea sunt:

Prototipul metodei	Descrierea metodei
void windowActivated (<code>WindowEvent e</code>)	Este invocată când fereastra este activată.
void windowClosed (<code>WindowEvent e</code>)	Este invocată după execuția metodei <code>dispose()</code> .
void windowClosing (<code>WindowEvent e</code>)	Este invocată când fereastra se dorește a fi închisă de către utilizator.
void windowDeactivated (<code>WindowEvent e</code>)	Este invocată când fereastra nu mai este activă.
void windowDeiconified (<code>WindowEvent e</code>)	Este invocată când fereastra trece din starea minimizată în starea normală.
void windowIconified (<code>WindowEvent e</code>)	Este invocată când fereastra trece din starea normală în starea minimizată.
void windowOpened (<code>WindowEvent e</code>)	Este invocată la prima afișare pe ecran a ferestrei.

Pentru cazul în care programatorul dorește să definească o clasă care să implementeze această interfață, vom extinde clasa `WindowAdapter`. Aceasta implementează interfața în cauză și implicit nu are nici un efect. Pentru a da anumite funcționalități programului în urma apariției anumitor evenimente, vom redefini metodele dorite. De exemplu, dacă dorim doar să facem posibilă închiderea ferestrei, vom redefini doar metoda `windowClosing()`. Avantajul constă în faptul că nu trebuie să mai implementăm toate metodele.

Exemplul 7.6.1. Implementarea interfeței `WindowListener` prin intermediul clasei `WindowAdapter`:

```
import java.awt.Frame;
import java.awt.event.*;
```

```
public class FereastraAscultator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        Frame fer = (Frame) e.getSource();
        fer.dispose();
    }
}
```

7.6.2. Clasa `java.awt.Frame`

Clasa `Frame` corespunde ferestrelor aplicațiilor. Aceasta este derivată din clasa `Window` și moștenește metodele sale, o parte din ele fiind descrise în secțiunea precedentă.

Cei mai utilizați constructori sunt cel implicit și cel cu un parametru de tip `String` care stabilește titlul ferestrei (cel care apare în bara ferestrei). Cele mai importante metode publice din clasa `Frame` sunt următoarele:

Prototipul metodei	Descrierea metodei
static <code>Frame[]</code> getFrames()	Returnează un vector cu toate obiectele de clasă <code>Frame</code> create de aplicație.
<code>MenuBar</code> getMenuBar()	Întoarce meniul de tip bară al aplicației.
void setMenuBar (<code>MenuBar</code> <code>menuBar</code>)	Stabilește meniul de tip bară al aplicației.
<code>String</code> getTitle()	Returnează titlul ferestrei.
void setTitle (<code>String</code> <code>titlu</code>)	Schimbă titlul ferestrei.
boolean isResizable()	Indică dacă fereastra se poate redimensiona.
void setResizable (boolean <code>b</code>)	Stabilește dacă fereastra poate fi redimensionată.

O aplicație care conține o fereastră principală va avea de cele mai multe ori clasa principală derivată din clasa `Frame`. Gestionarul de poziționare implicit este `BorderLayout`.

Exemplul 7.6.2. O aplicație simplă de sine stătătoare.

```
import java.awt.*;

public class AplicatieSimpla extends Frame {
    public AplicatieSimpla() {
        // apelarea constructorului superclasei
        super("AplicatieSimpla");
        // stabilirea ascultatorului pentru fereastra
        addWindowListener(new FereastraAscultator());
        // stabilirea pozitiei pe ecran si a dimensiunii
        setBounds(100,50,200,100);
        // crearea si adaugarea unei etichete
        Label eticheta = new Label(
```

```

        "Prima aplicatie de sine statatoare.");
        add(eticheta);
    }

    public static void main(String[] args) {
        AplicatieSimpla app = new AplicatieSimpla();
        // afisarea ferestrei
        app.show();
    }
}

```

Clasa `FereastraAscultator` este cea definită în exemplul 7.6.1.

7.6.3. Clasa `java.awt.Dialog`

Clasa `Dialog` permite lucrul cu ferestre de dialog. Acestea nu pot fi maximizate sau minimizate. Gestionarul implicit de poziționare este tot `BorderLayout`. Clasa `Dialog` posedă următorii constructori importanți:

- `public Dialog(Dialog d)`
- `public Dialog(Dialog d, String titlu)`
- `public Dialog(Dialog d, String titlu, boolean modal)`
- `public Dialog(Frame f)`
- `public Dialog(Frame f, String titlu)`
- `public Dialog(Frame f, String titlu, boolean modal)`

unde `f` și `d` sunt ferestrele părinte, `titlu` este titlul ferestrei, iar `modal` indică dacă fereastra de dialog este modală sau nu. O fereastră este *modală* dacă o dată activată nu se va putea reactiva fereastra părinte decât după închiderea acesteia. O fereastră este *nemodală* dacă permite activarea ferestrei părinte fără ca aceasta să se închidă.

Clasa `Dialog` este derivată din `Window` și are în plus două metode importante care se referă la tipul ferestrei (modală sau nemodală):

- `public boolean isModal()`
- `public void setModal(boolean b)`

Prima metodă testează dacă fereastra este modală, iar a doua stabilește dacă este modală (dacă `b` are valoarea `true`) sau nu.

Exemplul 7.6.3. Aplicația creează o fereastră de dialog și o activează la apăsarea unui buton. Fereastra de dialog se „ascunde” la apăsarea butonului `Ok` sau la închiderea acesteia.

```

import java.awt.*;
import java.awt.event.*;

public class ExempluDialog extends Frame
    implements ActionListener {
    Dialog dialog;

```

```

public ExempluDialog() {
    // apel la constructorul superclasei
    super("Exemplu de Dialog");
    // stabilirea pozitiei si a dimensiunilor
    setBounds(100,200,250,150);
    // adaugarea unui ascultator pentru fereastra
    addWindowListener(new FereastraAscultator());
    // crearea unui buton
    Button b = new Button("Dialog");
    add("North",b);
    b.addActionListener(this);
    // crearea unei etichete
    Label l = new Label(
        "Apasati pe butonul Dialog!");
    add(l);
    // crearea ferestrei de dialog
    dialog = new Dialog(this, "Test");
    FereastraDialogAscultator ascultator =
        new FereastraDialogAscultator(dialog);
    dialog.addWindowListener(ascultator);
    dialog.setBounds(150,250,150,100);
    // crearea unui buton si adaugarea la dialog
    Button ok = new Button("Ok");
    ok.addActionListener(ascultator);
    dialog.add(ok);
}

public static void main(String[] args) {
    ExempluDialog app = new ExempluDialog();
    app.show();
}

public void actionPerformed(ActionEvent e) {
    dialog.show();
}

public class FereastraDialogAscultator
    extends WindowAdapter
    implements ActionListener {

    private Dialog dialog;

    public FereastraDialogAscultator(Dialog dialog) {

```

```

        this.dialog = dialog;
    }

    public void windowClosing(WindowEvent e) {
        dialog.hide();
    }

    public void actionPerformed(ActionEvent e) {
        dialog.hide();
    }
}

```

7.6.4. Clasa java.awt.FileDialog

Selectarea numelor subdirectoarelor și a numelor fișierelor se realizează cu o fereastră de dialog specială pentru fișiere. Această îi corespunde clasa `FileDialog` care este derivată din `Dialog`. Apariția exactă depinde de sistemul de operare.

Există trei constructori pentru clasa `FileDialog`:

- `public FileDialog(Frame f)`
- `public FileDialog(Frame f, String titlu)`
- `public FileDialog(Frame f, String titlu, int mod)`

unde parametrii au aceeași semnificație ca la constructorii clasei `Dialog`, care au fost prezentați în subcapitolul 7.6.3, excepție făcând parametrul `mod`, care indică dacă fereastra este utilizată pentru salvarea sau încărcarea fișierului (sau directorului). Pentru acest parametru clasa `FileDialog` pune la dispoziție doi membri publici și statici de tip `int`: `SAVE` pentru salvare și `LOAD` pentru încărcare.

Metodele publice cele mai semnificative din clasa `FileDialog` sunt următoarele:

Prototipul metodei	Descrierea metodei
<code>String getDirectory()</code>	Obține directorul selectat.
<code>void setDirectory(String director)</code>	Stabilește directorul implicit.
<code>String getFile()</code>	Obține numele fișierului selectat.
<code>void setFile(String fisier)</code>	Stabilește un nume de fișier implicit.
<code>int getMode()</code>	Returnează tipul ferestrei: de încărcare sau de salvare.
<code>void setMode(int mod)</code>	Stabilește dacă fereastra este pentru încărcare sau pentru salvare.
<code>FilenameFilter getFilenameFilter()</code>	Returnează filtrul stabilit pentru fișiere.
<code>void setFilenameFilter(FilenameFilter filtru)</code>	Stabilește un filtru pentru fișierele ce vor apare în fereastră.

Pentru a stabili un filtru trebuie să construim o clasă care să implementeze interfața `FilenameFilter`. Aceasta face parte din pachetul `java.io` și conține o singură metodă care testează dacă un anumit fișier dintr-un anumit director este acceptat sau nu:

- `public boolean accept(File dir, String fisier)`

Exemplul 7.6.4. Caseta de dialog pentru fișiere apare când utilizatorul apasă butonul `Load`, apoi numele fișierului selectat este scris într-un câmp text.

```

import java.io.*;
import java.awt.*;
import java.awt.event.*;

public class ExempluFileDialog extends Frame
    implements ActionListener {

    private Button butonIncarca;
    private FileDialog obtineFisier;
    private TextField campText;

    public ExempluFileDialog() {
        super("Exemplu de FileDialog");
        setLayout(new FlowLayout());
        setSize(645, 382);
        addWindowListener(new FereastrăAscultator());

        butonIncarca = new Button("Load");
        add(butonIncarca);
        butonIncarca.addActionListener(this);

        campText = new TextField(20);
        add(campText);
    }

    public static void main(String[] args) {
        ExempluFileDialog app = new ExempluFileDialog();
        app.show();
    }

    public void actionPerformed(ActionEvent event) {
        String numeFisier;
        // crearea ferestrei de dialog pentru incarcare fisier
        obtineFisier = new FileDialog(this, "Obține fisier",
            FileDialog.LOAD);
        // activarea ferestrei
    }
}

```

```

    obtineFisier.show();
    // obtinerea fisierului
    numeFisier = obtineFisier.getFile();
    // afisarea in campul text
    campText.setText(numeFisier);
}
}

```

7.7. Managerul de securitate

O aplicație Java care accesează resursele sistem, cum ar fi monitorul, sistemul de fișiere, firele de execuție, procesele și rețeaua, poate fi controlată dintr-un singur punct cu un manager de securitate. Clasa care implementează această funcționalitate în Java este `java.lang.SecurityManager`.

Platforma Java furnizează un mecanism implicit de securitate care este suficient pentru multe aplicații. Însă pentru aplicațiile Java care necesită încărcare explicită de clase, este nevoie să scriem propriul manager de securitate.

O instanță a clasei `SecurityManager` poate fi creată doar o dată în mediul de execuție Java. Apoi, orice acces către o resursă sistem fundamentală este filtrată prin metode specifice ale obiectului de tip `SecurityManager`. Prin intermediul acestui obiect specializat, putem implementa politici de securitate simple sau complexe pentru accesul la resurse individuale.

Când începe execuția unui program Java, se așteaptă instalarea unui obiect `SecurityManager`. Dacă nu este instalat nici un manager de securitate (se utilizează `null`), atunci este permis orice tip de acces, așa încât sistemul de execuție Java va face orice activitate la același nivel de acces ca și alte programe care rulează cu autoritate de utilizator. Dacă aplicația care rulează necesită un mediu sigur, atunci se poate instala un obiect `SecurityManager` apelând metoda `System.setSecurityManager()`. De exemplu, navigatorul Web Netscape Navigator instalează un obiect `SecurityManager` înaintea rulării oricărui applet Java.

Aplicațiile care doresc instalarea unui obiect `SecurityManager` trebuie să deriveze clasa abstractă `java.lang.SecurityManager`. Deși această clasă este abstractă, ea nu conține nici o metodă abstractă, singurul motiv fiind că implementarea implicită nu este foarte utilă. Implicit, fiecare metodă de securitate din clasa `SecurityManager` este implementată pentru a furniza cel mai strict nivel de securitate, adică un obiect `SecurityManager` implicit va respinge toate cererile.

Exemplul 7.7.1. Următorul program Java instalează cel mai restrictiv obiect `SecurityManager`, astfel:

```

class TataRau extends SecurityManager {}

public class AplicatiaUnu {
    public static void main(String args[]) {

```

```

        System.setSecurityManager(new TataRau());
        // nu avem nici un fel de acces la fișiere, rețea,
        // ferestre, etc.
        System.out.println("Afisam un text sigur!");
    }
}

```

Programul `AplicatiaUnu` de mai sus nu poate decât să citească din `System.in` și să scrie în `System.out`. Orice încercare de a citi sau scrie alte fișiere, de a accesa rețeaua sau chiar a deschide o fereastră va implica aruncarea unei excepții `SecurityException`.

În exemplul 7.7.1, s-a instalat un obiect `SecurityManager` prea dur (intitulat, de altfel, `TataRau`). O dată instalat, este imposibil să schimbăm obiectul `SecurityManager` în vreun fel, deoarece securitatea acestuia nu este dependentă de obiectul `SecurityManager` însuși, ci este construit deja în mediul de execuție Java.

Astfel, pentru a avea un obiect `SecurityManager` folositor, va trebui să supra-scriem metodele care sunt consultate pentru diferitele tipuri de acces ale unor resurse. Acestea nu se apelează, de obicei, de programator (deși se poate), ci de nucleul Java înaintea acordării permisiunii unui tip particular de acces. Iată câteva dintre cele mai importante metode publice ale clasei `SecurityManager`:

Prototipul metodei	Descrierea metodei
<code>void checkAcces (Thread fir)</code>	Verifică accesul la firul de execuție <code>fir</code> .
<code>void checkExit (int stare)</code>	Pentru execuția <code>System.exit (stare)</code> ;
<code>void checkExec (String comanda)</code>	Verifică execuția procesului <code>comanda</code> .
<code>void checkRead (String fisier)</code>	Verifică drepturile de citire din fișierul specificat.
<code>void checkWrite (String fisier)</code>	Verifică drepturile de scriere din fișierul specificat.
<code>void checkDelete (String fisier)</code>	Verifică drepturile de ștergere a fișierului specificat.
<code>void checkConnect (String gazda, int port)</code>	Verifică accesul unui socket la serverul <code>gazda</code> și la portul <code>port</code> .
<code>void checkListen (int port)</code>	Verifică dreptul de a crea un socket la portul precizat.
<code>void checkAccept (String gazda, int port)</code>	Verifică dreptul de acceptarea a unei conexiuni la <code>gazda</code> și adresă <code>port</code> .
<code>void checkPropertyAccess (String cheie)</code>	Verifică accesul la proprietățile sistem.
<code>boolean checkTopLevelWindow (Object fereastră)</code>	Verifică dreptul de creare a unei noi ferestre.

Dacă accesul nu este permis, acestea aruncă o excepție `SecurityException`. Din metodele de mai sus, `checkTopLevelWindow()` este diferită prin faptul că întoarce

true, indicând astfel permisiunea accesului. Dacă întoarce false, atunci iar este permis accesul, însă se va produce o bordură de avertisment care identifică o fereastră nesigură (de neîncredere). Unele metode aruncă excepția `NullPointerException` în cazul în care este transmis ca parametru valoare null.

Exemplul 7.7.2. Vom implementa un manager de securitate care va permite deschiderea în acces de citire doar a fișierelor care au extensia .java:

```
class ExtensieJava extends SecurityManager {
    public void checkRead(String s) {
        if (s.endsWith(".java")) return ;
        else throw new SecurityException(
            "Accesul la citire este permis doar fișierelor" +
            "cu extensia .java");
    }
}
```

O dată ce managerul `ExtensieJava` este instalat, orice încercare de a citi alte fișiere decât cele cu extensia .java va arunca o excepție `SecurityException` însoțită de mesajul din program. Toate celelalte metode de securitate sunt moștenite din clasa `SecurityManager`, deci vor fi restrictive.

Toate restricțiile pentru appleturi dintr-o aplicație care vizualizează appleturi (eng. *applet-viewer*) sunt verificate de managerul de securitate, inclusiv dacă i se permite unui cod nesigur (de neîncredere) să fie încărcat prin rețea în vederea execuției acestuia.

7.8. Concluzii

Acest capitol a debutat cu prezentarea appleturilor, cu specificarea modului de includere a acestora în paginile Web, transmiterea parametrilor, modul de funcționare, comunicarea dintre appleturile care sunt în același document Web, redarea clipurilor și a imaginilor. De altfel, s-a prezentat și modalitatea de creare a unei aplicații care să fie simultan applet și aplicație de sine stătătoare.

S-au prezentat aplicațiile conduse de evenimente, modalitatea de desenare, utilizarea componentelor și a containerelor, a elementelor de formular pentru introducerea facilă a datelor, tratarea evenimentelor prin intermediul ascultătorilor și adaptorilor. De asemenea, s-au prezentat gestionarii de poziționare a subcomponentelor unui container, precum crearea de ferestre și de aplicații de sine stătătoare prin utilizarea bibliotecii AWT.

S-a arătat modalitatea de stabilire a unei politici de restricții pentru aplicații prin intermediul managerului de securitate.

7.9. Test grilă

Întrebarea 7.9.1. Care dintre afirmațiile de mai jos referitoare la appleturi sunt adevărate?

- a) Sunt executate de către serverul Web.
- b) Sunt stocate pe serverul Web.
- c) Sunt compilate de navigatorul Web.
- d) Sunt executate de navigatorul Web.

Întrebarea 7.9.2. Precizați care dintre afirmațiile următoare sunt false:

- a) Orice navigator Web poate rula appleturi.
- b) Appleturile pot fi stocate pe orice server Web.
- c) Appleturile pot fi stocate local, pe calculatorul clientului.
- d) Cel mult un applet poate fi conținut de un document Web.

Întrebarea 7.9.3. Un applet poate fi:

- a) Executat cu `appletviewer` fără să creăm o pagină HTML.
- b) Simultan și aplicație de sine stătătoare.
- c) Executat cu ajutorul comenzii `java`.
- d) Compilat doar la încărcarea acestuia în navigatorul Web.

Întrebarea 7.9.4. Care dintre propozițiile următoare referitoare la metodele unui applet sunt adevărate?

- a) Trebuie să redefinim măcar o metodă, altfel obținem eroare la compilare.
- b) Sunt apelate automat de navigatorul Web.
- c) Pot fi apelate direct de către utilizatori.
- d) Nu se pot declara noi metode.

Întrebarea 7.9.5. Care dintre afirmațiile următoare sunt false?

- a) Doar appleturile din aceeași pagină Web pot comunica între ele.
- b) Un applet poate fi configurat din cadrul paginii Web.
- c) Un document Web poate conține un applet de pe alt server Web.
- d) În anumite situații un applet poate avea acces la sistemul de fișiere local.

Întrebarea 7.9.6. Un applet poate conține:

- a) imagini stocate în fișiere externe;
- b) elemente de redare a sunetelor aflate în fișiere externe;
- c) texte de diferite dimensiuni și culori;
- d) figuri geometrice bidimensionale;
- e) Nu poate conține simultan imagini, desene, text și audio.

Întrebarea 7.9.7. O componentă grafică trebuie neapărat să:

- a) conțină o reprezentare grafică;
- b) aibă atașat un ascultător pentru interceptarea acțiunilor;

- c) fie vizibilă;
- d) fie atașată unei suprafețe de desenare.

Întrebarea 7.9.8. Care dintre următoarele clase nu reprezintă componente grafice?

- a) Panel
- b) Canvas
- c) SoundComponent
- d) Container
- e) Applet
- f) List

Întrebarea 7.9.9. Care dintre gestionarii de poziționare de mai jos pot fi utilizați de appleturi?

- a) GridBagLayout
- b) TableLayout
- c) GroupLayout
- d) FlowLayout
- e) WindowLayout

Întrebarea 7.9.10. Care dintre următoarele metode trebuie implementate pentru tratarea evenimentului de apăsare a unui buton?

- a) Nu trebuie definită nici o metodă.
- b) actionPerformed()
- c) ActionListener()
- d) itemStateChanged()

Întrebarea 7.9.11. Care dintre următoarele clase definesc ferestre?

- a) FileDialog
- b) WindowListener
- c) Applet
- d) Frame
- e) Applet
- f) MainWindow

Întrebarea 7.9.12. Un manager de securitate poate stabili:

- a) drepturile de acces pentru fișiere;
- b) restricții pentru executarea anumitor aplicații;
- c) care appleturi să poată accesa sistemul local de fișiere;
- d) doar restricții, nu și drepturi.

Întrebarea 7.9.13. Ce metodă trebuie definită pentru a putea desena pe suprafața appletului?

- a) Nu trebuie definită nici o metodă.
- b) update()
- c) paint()
- d) repaint()

7.10. Exerciții propuse spre implementare

Exercițiul 7.10.1. Presupunem că suntem în campanie electorală și dorim să vizualizăm situația statistică a unui sondaj electoral. Presupunem că avem următorul caz:

- partidul P1 42%
- partidul P2 12%
- partidul P3 8%
- partidul P4 6%
- partidul P5 5%
- celelalte partide 27%

Realizați un applet care să pună în evidență aceste proporții. De exemplu, un cerc care să conțină diferite culori pentru anumite sectoare de cerc, reprezentate proporțional cu procentele furnizate în urma sondajului electoral.

Exercițiul 7.10.2. Scrieți un program Java care să convertească mile în kilometri (și invers).

Exercițiul 7.10.3. Construiți un applet care să permită micșorarea/mărirea lungimii catetelor unui triunghi dreptunghic folosind două bare de defilare. De asemenea, afișați și tangenta unui unghi ascuțit ($tg(\alpha) = \frac{cateta1}{cateta2}$).

Exercițiul 7.10.4. Scrieți un applet care să afișeze 10 butoane corespunzătoare cifrelor zecimale și un buton corespunzător trimerii datelor și validării ghicirii combinației de cifre secrete setate prin program. (Este vorba, de fapt, de o parolă la începutul execuției programului.)

Exercițiul 7.10.5. Realizați jocul X și O. Se va da posibilitatea de a juca utilizatorul cu calculatorul.

Exercițiul 7.10.6. Utilizând componenta TextArea și un buton, scrieți un program Java care calculează numărul de caractere și de linii din text.

Exercițiul 7.10.7. Scrieți un program Java care creează o arie text și butoane pentru operațiile de cut-copy-paste (decupare, copiere și inserare).

Exercițiul 7.10.8. Construiți un program Java care are două butoane și un TextField pentru citirea unui fișier grafic sau audio. Butoanele sunt corespunzătoare apelurilor getImage() și getAudioClip(). Programul trebuie să citească numele fișierului și să-l afișeze (să-l interpreteze).

Exercițiul 7.10.9. Definiți un Frame care generează un alt Frame mai mic, care la rândul său generează alt Frame și mai mic, și tot așa (maximum 6 iterații). Fiecare Frame are un buton a cărui apăsare implică generarea următorului Frame.

Exercițiul 7.10.10. Creați un applet care permite inserarea unei adrese a unei pagini Web. La acționarea unui buton se va accesa pagina indicată. Se va da posibilitatea utilizatorului, printr-un buton radio, să aleagă dacă pagina va fi afișată în aceeași fereastră a navigatorului Web sau într-una nouă.

7.11. Proiecte propuse spre implementare

Proiectul 7.11.1 Scrieți un program Java (care să fie simultan applet și aplicație de sine stătătoare) care să simuleze un calculator de buzunar.

Proiectul 7.11.2. Să se construiască o aplicație care permite crearea de rebusuri. Fiecare rebus va fi salvat într-un fișier separat și va putea fi utilizat de proiectul 7.11.3.

Proiectul 7.11.3. Să se realizeze o aplicație care să permită rezolvarea de rebusuri. Prin intermediul unei liste de opțiuni se va alege rebusul dorit. Fiecare rebus este definit într-un fișier creat de proiectul precedent.

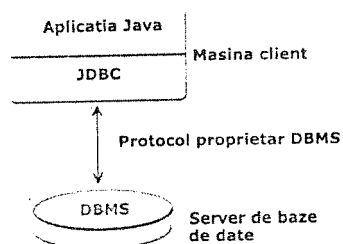
8. Accesul la baze de date folosind JDBC

8.1. Cuvinte cheie

- baze de date
- DBMS (DataBase Management System)
- drivere JDBC
- interogări SQL
- mulțimi rezultat ResultSet
- tranzacții

8.2. Introducere

JDBC (eng. *Java DataBase Connectivity*) reprezintă API-ul (eng. *Application Programming Interface*) dezvoltat de Sun Microsystems în colaborare cu diverși parteneri pentru a oferi aplicațiilor Java acces spre bazele de date gestionate de diverse DBMS-uri (Sisteme de Gestionare a Bazelor de Date). Într-o arhitectură client-server, bazele de date se pot afla pe aceeași mașină sau pe o altă mașină cu care clientul este conectat dintr-un intranet sau chiar Internet. Cea mai importantă funcție a JDBC-ului este posibilitatea lucrului cu instrucțiuni SQL (eng. *Structured Query Language*) și procesarea eventualelor rezultatelor obținute în urma interogărilor într-o manieră independentă și consistentă. Putem privi API-ul JDBC ca o interfață standard între aplicații și diverse DBMS-uri, creatorii aplicațiilor programând conform acestui API într-un mod uniform și independent de DBMS. Urmează ca apelurile SQL să fie preluate, traduse și trimise mai departe de diverse drivere scrise în parte pentru fiecare DBMS.



Așa cum se poate vedea în figura precedentă, prin introducerea unui nivel de abstractizare reprezentat de API-ul JDBC (această metodă de lucru este des întâlnită în limbajul Java), se evită situația neplăcută ca programatorii să gestioneze singuri apelurile SQL către un DBMS particular, în acest caz codul aplicației trebuind modificat pentru a se putea accesa un alt DBMS. Prin introducerea JDBC, pentru a ne conecta la un alt DBMS, este îndeajuns să schimbăm driverul, operație care se poate realiza dinamic, chiar în timpul rulării aplicației, nemaifiind nevoie de recompilarea acesteia. Această facilitate, împreună cu portabilitatea limbajului Java, permite interogarea oricărei baze de date de sub orice platformă fără recompilări de cod.

Modul în care driverele trebuie construite este standardizat prin intermediul specificațiilor JDBC care specifică interfețele standard care trebuie implementate de cei care construiesc drivere. De-a lungul timpului s-a înregistrat o evoluție a acestor specificații în sensul creșterii funcționalităților pe care driverele le pun la dispoziție utilizatorilor, fără a afecta compatibilitatea cu versiunile anterioare ale specificațiilor. În general, specificațiile JDBC descriu o serie de interfețe pe care cei ce implementează driverele trebuie să le implementeze. Unele DBMS-uri nu permit folosirea procedurilor stocate sau a altor principii din cauza dezvoltării într-un mod nestandard a pieței bazelor de date. De aceea, specificațiile JDBC apărute de-a lungul timpului îi obligă pe cei care creează drivere la implementarea unei liste restrânse de interfețe

(eventual numai anumite metode ale lor), implementarea altora rămânând opțională, nerestricționând astfel posibilitățile reale ale DBMS-urilor deja existente.

Începând cu specificația JDBC 2.0, API-ul este împărțit în două părți: JDBC 2.1 API, care reprezintă nucleul API-ului JDBC, cuprins în pachetul `java.sql`, și JDBC 2.0 Optional Package, care oferă capabilități pe partea de server cum ar fi tranzacții distribuite și conexiuni multiple, fiind conținut în pachetul `javax.sql`. Ultima specificație JDBC existentă în formă finală (eng. *final release*) are versiunea 3.0 și o puteți găsi la adresa `java.sun.com/products/jdbc`. Nu aduce diferențe majore față de precedenta versiune 2.0, ci o prezintă pe aceasta într-o formă compactă și completând unele aspecte legate de aplicațiile distribuite. De aceea, cuprinsul acestui capitol corespunde nucleului versiunii 2.0. Această specificație aduce multe îmbunătățiri față de mai vechea distribuție 1.0, cum ar fi folosirea cursorilor, suport pentru stocarea obiectelor, acces programatic la bazele de date.

Pentru început putem spune că JDBC furnizează acces orientat pe obiecte (OOP) la bazele de date prin definirea de clase și interfețe care înfășoară diverse concepte abstracte, cum ar fi cele prezentate în tabelul următor:

Concepte înfășurate	Clasa/clasele sau interfețele corespunzătoare
Conexiuni la baze de date	Connection
Interogări SQL	Statement, PreparedStatement, CallableStatement
Mulțimi rezultat	ResultSet
Obiecte mari binare sau caracter	Blob (eng. Binary Large Objects) Clob (eng. Character Large Objects)
Drivere	Driver
Gestionari de drivere	DriverManager

De asemenea, standardul JDBC definește o serie de interfețe care trebuie implementate de creatorii driverelor cu scopul de a oferi dezvoltatorilor informații despre baza de date interogată, DBMS-ul folosit etc. Vom numi aceste informații *metadate* în sensul de „date despre date”.

Interfețele puse la dispoziție pentru obținerea metadatelor le vom folosi în exemplele acestui capitol, și sunt prezentate în tabelul care urmează:

Interfață	Descriere
DatabaseMeta Data	Interfață folosită de către cei care produc drivere pentru a informa utilizatorii despre capabilitățile oferite de DBMS-ului împreună cu driverul JDBC folosit.
ParameterMeta Data	Interfață folosită pentru a obține informații despre tipurile și proprietățile parametrilor dintr-un obiect <code>PreparedStatement</code> .
ResultSetMeta Data	Interfață folosită pentru a obține informații despre tipurile și proprietățile coloanelor dintr-un <code>ResultSet</code> .

Ceea ce trebuie remarcat este faptul că aplicațiile care folosesc bazele de date trebuie să includă pachetul `java.sql`, și nu pachetul care conține implementarea

driverului particular folosit. Totuși, calea spre pachetul conținând driverul trebuie să fie prezentă în CLASSPATH.

8.3. Clasificarea driverelor JDBC

Sub aspectul funcționalității, există patru tipuri de drivere JDBC:

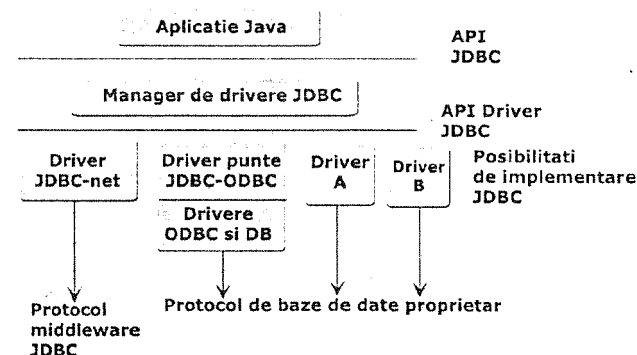
1. **Puntea JDBC-ODBC** – Acționează ca o legătură dintre JDBC și alt mecanism de conectivitate a bazelor de date numit ODBC (eng. *Object DataBase Connectivity*). Acest driver vine o dată cu ultimele distribuții ale platformelor Java și este reprezentat în Java 2 SDK de clasa `sun.jdbc.odbc.JdbcOdbcDriver`. Este folosit în aplicații de accesare a datelor unde nu există drivere JDBC pure. Puntea traduce metodele JDBC în apeluri de funcții ODBC. Puntea JDBC-ODBC necesită ca bibliotecile ODBC native și driverele ODBC să fie instalate și configurate pentru fiecare client ce folosește un driver de tip 1. Această cerință reprezintă o limitare serioasă pentru multe aplicații, funcționând doar pentru sistemele de operare *Microsoft Windows* și *Sun Solaris*. Mai multe informații puteți găsi în secțiunea acestui capitol dedicată utilizării punții JDBC-ODBC.
2. **Java-API nativ** – Aceste drivere folosesc interfața nativă Java (eng. *JNI – Java Native Interface*) pentru a face apeluri direct la API-ul unei baze de date locale. De obicei, driverele de tip 2 sunt mai rapide decât cele de tip 1. Ca și driverele de tip 1, driverele de tip 2 necesită instalarea și configurarea bibliotecilor client bază de date native pe mașina client. Sunt foarte convenabile când există biblioteci de accesare a datelor scrise în limbajul C, însă acestea nu sunt portabile pe toate platformele. Driverele de tip 2 sunt dedicate unui singur DBMS.
3. **Java-protocol de rețea** – Acest tip de drivere sunt drivere Java pure care folosesc un protocol de rețea (de obicei, TCP/IP) pentru a comunica cu aplicația JDBC *middleware*. Apoi, aplicația JDBC *middleware* traduce cererile JDBC folosind protocolul de rețea în apeluri de funcții specifice bazelor de date. Tipul 3 de drivere reprezintă soluția cea mai flexibilă, deoarece nu necesită biblioteci de bază de date native pe client și se poate conecta la mai multe baze de date diferite.
4. **Java-protocol bază de date** – Acest tip de drivere sunt drivere Java pure care implementează un protocol de bază de date (cum ar fi *Oracle SQL Net*) pentru a comunica direct cu baza de date. De aceea, aceste drivere sunt cele mai rapide. Ca și driverele de tip 3, ele nu necesită biblioteci de bază de date native și pot fi folosite distribuit fără instalarea clientului. Driverele de tip 4 sunt specifice unui singur DBMS. Mai multe informații puteți găsi în secțiunea instalarea și configurarea MySQL.

În general, driverele de tip 1 și 2 sunt gratuite, însă necesită instalarea la nivelul client a bibliotecile ODBC native, fiind folosite doar acolo unde soluții dezvoltate

integral în Java nu sunt încă disponibile. Driverele de tip 3 sunt, în general, comerciale și se pot folosi în aplicații flexibile pe mai multe platforme. Driverele de tip 4 sunt mai rapide, unele sunt chiar gratis, însă se pot folosi doar pentru DBMS-uri particulare. Driverele de tipul 3 și 4 oferă toate avantajele tehnologiei Java, printre care posibilitatea download-ului acestora în timp ce aplicația este folosită.

Puteți afla lista driverelor JDBC din cele patru tipuri pentru diverse DBMS-uri și conformanța lor cu diversele specificații existente pe situl firmei SUN la adresa `industry.java.sun.com/products/jdbc/drivers`. Se observă faptul că distribuțiile DBMS neproprietare oferă, în general, suport mai scăzut pentru Java decât cele proprietare, care au în general drivere conforme chiar cu specificația 3.0.

Pentru gestiunea driverelor folosite într-o aplicație, JDBC API folosește un manager de drivere reprezentat de o instanță a clasei `DriverManager`. Managerul de drivere este capabil să suporte drivere concurente multiple conectate la baze de date eterogene multiple. În figura care urmează sunt prezentate aspecte ale conectării la baze de date:



8.4. Breviar SQL

SQL (eng. *Structured Query Language*) a devenit standard oficial pentru accesul la baze de date relaționale în 1980. Motivul acceptării SQL ca limbaj de interogare relațional a fost în principal apariția arhitecturilor client/server. În ceea ce privește interogările SQL, orice driver JDBC trebuie să suporte cel puțin secțiunea Entry Level a standardului adoptat în 1992 de ANSI, care poartă numele *SQL-92* (Entry Level reprezintă o serie de funcționalități SQL specificate de acest standard). Trebuie remarcat faptul că driverele conforme JDBC 2.0 oferă suport și pentru tipuri *SQL3*.

Instrucțiunile SQL se împart în două categorii:

1. Instrucțiuni ale limbajului de definire a datelor (eng. *DDL – Data Definition Language*). Sunt folosite pentru crearea și modificarea obiectelor bazei de date;
2. Instrucțiuni ale limbajului de manipulare a datelor (eng. *DML – Data Manipulation Language*). sunt folosite pentru operații asupra datelor unei baze de date.

Acestea pot fi împărțite în două categorii:

- 2.1. Instrucțiunea SELECT – care întoarce o mulțime de date rezultat ce satisfac un criteriu de interogare;
- 2.2. Alte instrucțiuni – care nu întorc o mulțime rezultat.

Pentru crearea unei tabele (structura unei baze de date), vom folosi DDL, care definește sintaxa pentru comenzile: CREATE TABLE și ALTER TABLE.

Pentru adăugarea, modificarea, ștergerea, căutarea datelor într-o bază de date se folosește limbajul DML (instrucțiunile INSERT, UPDATE, DELETE, SELECT).

Prezentăm în continuare o descriere amănunțită a principalelor instrucțiuni SQL, care vor fi folosite pe parcursul acestui capitol. Pentru un tutorial SQL, puteți consulta www.w3schools.com/sql.

8.4.1. Instrucțiunea CREATE TABLE

Sintaxa generală a unei comenzi CREATE TABLE este:

```
CREATE TABLE <numetabela> (
    <numecamp> <tipdatasql> <descrieretip>,
    <numecamp> <tipdatasql> <descrieretip>,
    . . .
    <numecamp> <tipdatasql> <descrieretip>
);
```

<tipdatasql> poate fi: char, varchar, boolean, smallint, integer, numeric, float, currency, double, date, time, datetime, raw. <descrieretip> se referă la caracteristicile tipului de dată SQL și poate fi numărul maxim de caractere, dacă este cheie primară etc.

Exemplu:

```
CREATE TABLE studenti (
    nr int not NULL auto_increment primary key,
    nume varchar(25) NULL,
    prenume varchar(25) NULL,
    nota int NULL
);
```

8.4.2. Instrucțiunea INSERT

Definirea unei instrucțiuni INSERT implică trei pași:

1. definirea tabelului de destinație pentru inserarea datelor;
2. definirea câmpurilor (coloanelor) care vor primi valori;
3. definirea valorilor pentru aceste coloane.

Astfel, sintaxa generală este:

```
INSERT INTO <numetabela> (camp1, camp2, . . . , campn)
VALUES (valcamp1, valcamp2, . . . , valcampn);
```

Acțiunea acestei instrucțiuni SQL va fi adăugarea unei înregistrări ale cărei câmpuri sunt inițializate cu valorile precizate biunivoc.

Exemplu:

```
INSERT INTO studenti (nr, nume, prenume, nota) VALUES (1,
Popescu, Ion, 7);
INSERT INTO studenti (nume, prenume) VALUES (Ionescu, Ion);
// studentul Ionescu Ion va primi nota mai tarziu !
```

Dacă se dorește completarea tuturor câmpurilor unei înregistrări, atunci se poate omite partea a doua, adică lista câmpurilor. Astfel, putem rescrie instrucțiunea de mai sus:

Exemplu: INSERT INTO studenti VALUES (1, Popescu, Ion, 7);

8.4.3. Instrucțiunea UPDATE

Definirea unei instrucțiuni UPDATE necesită trei pași:

1. definirea tabelului de destinație pentru modificarea datelor;
2. definirea câmpului și a unei noi valori;
3. definirea filtrului tabelului.

Astfel, sintaxa generală este:

```
UPDATE <numetabela> SET camp1 = valcamp1 WHERE camp2 =
valcamp2;
```

Acțiunea acestei instrucțiuni SQL va fi modificarea (înlocuirea) unei valori a unui câmp dintr-o înregistrare.

Exemplu: UPDATE studenti SET nota = 8 WHERE nr = 2;

8.4.4. Instrucțiunea DELETE

Definirea unei instrucțiuni DELETE necesită doi pași:

1. definirea tabelului de destinație pentru ștergerea datelor;
2. definirea filtrului tabelului.

Astfel, sintaxa generală este:

```
DELETE FROM <numetabela> WHERE <conditie>;
```

Acțiunea acestei instrucțiuni SQL va fi ștergerea unor înregistrări.

Exemplu: DELETE FROM studenti WHERE nr = 2;

8.4.5. Instrucțiunea SELECT

Se utilizează pentru a obține înregistrări ale bazei de date care îndeplinesc anumite criterii. Există patru părți ale acestei instrucțiuni:

1. definirea a ceea ce dorim să obținem;
2. definirea tabelului de unde se vor obține informațiile;
3. definirea condițiilor de obținere (tabele join, filtre pentru înregistrări);
4. definirea ordinii în care se dorește vizualizarea datelor.

Rezultatul întors de instrucțiunea SELECT este o mulțime de înregistrări (eng. *ResultSet*). Așadar, un *ResultSet* poate fi privit ca o tabelă de date cu număr fixat de linii și coloane.

Sintaxa generală a unei instrucțiuni select este:

```
SELECT <listacampuri> [AS <alias>] FROM <numetabela> WHERE
<expresiebooleana>;
```

În vederea obținerii unei select cu JOIN (submulțime a unui produs cartezian), partea <listacampuri> depinde de <numebazadate>.

Exemple:

```
// obtinerea listei studentilor promovati
SELECT nume, prenume AS porecla FROM studenti WHERE nota > 5;
// daca se doreste selectarea tuturor inregistrarilor,
// atunci se foloseste '*'
SELECT * FROM studenti;
// presupunem ca avem trei baze de date specifice bibliotecii facultatii
// de informatica. o baza se numeste autori cu structura (nr, nume,
// prenume), alta baza se numeste carti cu structura (isbn, titlu,
// bucati) si ultima baza de date se numeste autoricarti cu structura
// (nr, isbn). Dorim sa selectam o lista de autori si cartile lor
// corespunzatoare.
SELECT a.nume, a.prenume, b.titlu FROM autori a, carti b,
autoricarti ab WHERE a.nr = ab.nr AND b.isbn = ab.isbn;
```

8.5. Mapări de tipuri între SQL și Java

Tipurile de date din SQL sunt diferite de tipurile de date suportate de limbajul Java. De asemenea, există deosebiri semnificative între tipurile suportate de diferite DBMS-uri existente pe piață. Sau chiar dacă practic acestea suportă tipuri cu aceeași semnificație, tipurile poartă alte nume. Spre exemplu, tipul *BYTE* din Acces 7.0 corespunde tipului *TINYINT* din Sybase 11.9 și ambele se referă la un întreg din intervalul [0,255], eventual cu semn. Pentru a evita aceste incompatibilități, JDBC definește o serie de

tipuri generice SQL prin intermediul clasei *java.sql.Types*, ceea ce permite ca programatorul să nu se îngrijească de numele tipurilor de date folosite de DBMS-ul la care se conectează. Trecerea între tipurile generice JDBC și tipurile efective caracteristice bazelor de date cade în sarcina driverului. Programatorul interacționează în aplicația lui numai cu tipurile JDBC, fiind definită o mapare standard între tipurile JDBC și tipurile Java. Acest lucru determină o interfață simplă pentru scrierea și citirea de valori ca simple tipuri Java. Tabelul următor prezintă maparea JDBC-Java:

Tip JDBC	Tip Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	Clob
BLOB	Blob
ARRAY	Array
DISTINCT	tip corespunzator
STRUCT	Struct
REF	Ref
JAVA_OBJECT	clasa JAVA corespunzatoare

Un programator trebuie să țină cont de acest tabel atunci când scrie aplicația care va interacționa cu baza de date. Spre exemplu, dacă în unul dintre tabelele interogate avem un câmp de tip *REAL*, atunci va trebui să folosească o variabilă de tip *float* pentru a ține informațiile din celulele acelei coloane.

Locul în care programatorii interacționează totuși cu tipuri de date folosite de DBMS îl reprezintă instrucțiunile SQL de tip *CREATE TABLE*, pe care aceștia le folosesc pentru a crea tabele. În acest caz, este indicată citirea cu atenție a documentației DBMS-ului. O modalitate de a scrie totuși comenzi *CREATE TABLE* care să

funcționeze pentru o varietate de baze de date este folosirea unui număr redus de tipuri de date, cum ar fi `INTEGER`, `NUMERIC` sau `VARCHAR`, unanim acceptate de toate DBMS-urile. O altă metodă ar fi folosirea metodei `java.sql.DatabaseMetaData.getTypeInfo()` pentru a descoperi tipurile de date acceptate de DBMS.

În continuare, prezentăm mecanismele pentru transferarea datelor între aplicații care folosesc tipuri Java și bazele de date interogate care folosesc tipuri SQL. Aceste metode le veți regăsi folosite în secțiunile următoare.

Metode interacțiune	Forma generică	Observații
Metode aplicate <code>ResultSet</code> -ului obținut în urma executării interogărilor SQL.	Metode de forma <code>getXXX</code> unde <code>XXX</code> reprezintă tipul de date preferat. În cazul JDBC 2.0 apar și metode de forma <code>updateXXX()</code>	Mai general, metoda <code>getString()</code> poate fi folosită pentru a returna orice tip de date cu precizarea că cele care aveau tipul numeric trebuie reconvertite. De asemenea, în același scop se pot folosi apeluri <code>getObject()</code> cu operatorii cast corespunzători.
Metode aplicate instanțelor <code>PreparedStatement</code> pentru a da valori instrucțiunilor parametrizate.	Metode de forma <code>setXXX</code> unde <code>XXX</code> reprezintă tipul de date preferat.	<code>XXX</code> reprezintă un tip Java oarecare.
Metode aplicate instanțelor <code>CallableStatement</code> pentru a obține parametrii de ieșire.	Metode de forma <code>getXXX</code> unde <code>XXX</code> reprezintă tipul de date preferat.	Se poate folosi, de asemenea, <code>getObject()</code> , împreună cu operatorii cast corespunzători.

JDBC 2.0 aduce noi tipuri de date cunoscute ca tipuri SQL3. Corespunzător acestor tipuri, JDBC oferă interfețe pentru maparea lor. Prezentăm în continuare tipurile SQL, descrierea lor și interfețele JDBC corespunzătoare:

Tip SQL3	Interfață JDBC	Descriere
BLOB	Blob	Oferă posibilitatea stocării unei cantități mari de date reprezentate sub formă de șir de biți (binară).
CLOB	Clob	Oferă posibilitatea stocării unei cantități foarte mari de date în format caracter.
ARRAY	Array	Se pot stoca tablouri.
Tip structurat SQL	Struct	Permite folosirea tipurilor de date structurate SQL definite de utilizator.
REF	Ref	Permite accesarea prin referință a tipurilor de date structurate SQL definite de utilizator.

Pentru a citi, stoca și actualiza date de tip SQL3, se vor folosi metode de forma `getXXX`, `setXXX` și `updateXXX`, unde `XXX` reprezintă tipul corespunzător coloanei a doua din tabelul de mai sus.

SQL permite și definirea de tipuri de date utilizator, cunoscute și sub numele de UDT (eng. *User Defined Types*) prin intermediul instrucțiunilor SQL `CREATE TYPE`. Acestea se împart în tipuri structurate și tipuri distincte. Spre exemplu, prezentăm modul de creare a unui tip structurat de date:

```
CREATE TYPE PUNCT_PLAN
(
    X FLOAT,
    Y FLOAT
)
```

Un tip distinct poate fi creat având la bază un tip deja existent, așa cum se poate vedea în exemplul următor:

```
CREATE TYPE NUMERE AS NUMERIC(10, 2)
```

Prin definiție, un tip distinct SQL este mapat la același tip ca și tipul de bază. În exemplul prezentat, deoarece tipul `NUMERIC` este mapat la `java.math.BigDecimal`, aceeași mapare va căpăta și tipul `NUMERE`. Deci vom folosi `ResultSet.getBigDecimal()` pentru a obține o valoare de acest tip.

8.6. Accesarea unei baze de date folosind JDBC

Procesul obținerii de informații dintr-o bază de date folosind JDBC implică în principiu cinci pași:

1. înregistrarea driverului JDBC folosind gestionarul de drivere `DriverManager`;
2. stabilirea unei conexiuni către baza de date;
3. execuția unei instrucțiuni SQL;
4. procesarea rezultatelor;
5. închiderea conexiunii cu baza de date.

8.6.1. Înregistrarea driverului JDBC folosind clasa `DriverManager`

Funcționalitatea managerului de drivere este aceea de a menține o referință către toate obiectele driver disponibile în aplicația curentă. Un driver JDBC este înregistrat automat de managerul de drivere atunci când clasa driver este încărcată dinamic. Pentru încărcarea dinamică a unui driver JDBC, folosim metodele `Class.forName()`. După cum se observă, nu este nevoie de crearea unei instanțe a clasei driver o dată ce aceasta a fost încărcată. Dacă se apelează metoda `newInstance()`, se va realiza un duplicat nefolositor al clasei driver. În exemplul următor vom încărca driverul punte JDBC-ODBC din pachetul `sun.jdbc.odbc` și, mai apoi, driverul MySQL Connector/J, care permite conectarea la serverul de baze de date MySQL. Trebuie remarcat faptul că pentru Connector/J este necesar să includem în `CLASSPATH` calea spre pachetul care conține driverul.

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("com.mysql.jdbc.Driver");
```

`Class.forName()` este o metodă statică ce permite mașinii virtuale Java să aloce dinamic, să încarce și să facă o legătură la clasa specificată ca argument printr-un șir de caractere. În cazul în care clasa nu este găsită, se aruncă o excepție `ClassNotFoundException`. A se vedea Java Reflection API.

Driverurile pot fi înregistrate și folosind metoda `DriverManager.registerDriver()`.

8.6.2. Stabilirea unei conexiuni către baza de date

O dată ce s-a încărcat un driver, putem să-l folosim pentru stabilirea unei conexiuni către baza de date. O conexiune JDBC este identificată printr-un URL JDBC specific. Sintaxa standard pentru URL-ul unei baze de date este:

```
jdbc:<subprotocol>:<nume>
```

Prima parte precizează că pentru stabilirea conexiunii se folosește JDBC. Partea de mijloc `<subprotocol>` este un nume de driver valid sau al altei soluții de conexiune a bazelor de date. Ultima parte, `<nume>`, este un nume logic sau alias care corespunde bazei de date fizice. Dacă baza de date va fi accesată prin Internet, secțiunea `<nume>` va respecta următoarea convenție de nume:

```
//numegazda:port/subsubnume
```

În acest sens, o adresă corectă este următoarea:

```
jdbc:mysql://localhost:386/arhiva
```

Prezentăm în continuare sintaxa particulară pentru câteva drivere JDBC:

```
ODBC - jdbc:odbc:<sursa_date>[;<nume_tribut>=<valoare_tribut>]*
MySQL - jdbc:mysql://server[:port]/numeBazaDate
Oracle - jdbc:oracle:thin:@server:port:numeInstanta
```

Pentru stabilirea unei conexiuni la o bază de date, se folosește metoda statică `getConnection()` din clasa `DriverManager`.

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/arhiva");
```

Pentru baze de date care necesită autentificare, se utilizează o formă a acestei metode cu trei argumente.

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/arhiva", nume_utilizator, parola);
```

Pentru a afla informații despre DBMS, va trebui să apelăm `Connection.getMetaData()` pentru a obține o instanță `DatabaseMetaData`, căreia îi putem aplica diverse metode a afla informații despre tabelele bazei de date, gramatica SQL suportată, dacă suportă sau nu proceduri stocate etc.

Folosind JNDI (Java Name and Directory Interface) putem să ne conectăm la o bază de date considerând-o ca sursă de date având un nume logic, în loc să codăm

direct în aplicație numele ei și driverul pe care îl vom folosi. Acest mod de conectare nu face subiectul prezentului capitol, fiind o caracteristică a sistemelor distribuite.

8.6.3. Execuția unei instrucțiuni SQL

După ce s-a stabilit conexiunea, se pot trimite instrucțiuni SQL către baza de date. API-ul JDBC nu verifică corectitudinea instrucțiunii și nici apartenența ei la un anumit standard SQL, permițându-se astfel trimiterea chiar de instrucțiuni non-SQL. Programatorul este cel care știe dacă DBMS-ul interogă suportă interogările pe care le trimite și, dacă nu, el este cel care va trata excepțiile primite drept răspuns. Dacă instrucțiunea este SQL, atunci aceasta poate face anumite operații asupra bazei de date, cum ar fi căutare, inserare, actualizare sau ștergere.

API-ul JDBC specifică trei interfețe (cărora cel ce creează driverul trebuie să le dea implementare) pentru trimiterea de interogări către bazele de date, fiecareia corespunzându-i o metodă specială în clasa `Connection`, de creare a instanțelor corespunzătoare. Acestea sunt prezentate în tabelul care urmează:

Clasa	Metoda de creare	Explicații
Statement	<code>Connection.createStatement()</code>	Este folosită pentru trimiterea de instrucțiuni SQL simple, fără parametri.
PreparedStatement	<code>Connection.prepareStatement()</code>	Permite folosirea instrucțiunilor SQL precompilate și a parametrilor de intrare în interogări. Această metodă de a face interogări este utilă în cazul în care se fac interogări care diferă doar printr-un număr de parametri.
CallableStatement	<code>Connection.prepareCall()</code>	Permite folosire procedurilor stocate pe serverul DBMS.

Pentru execuția unei instrucțiuni SQL neparametrizate, se folosește metoda `createStatement()`, aplicată unui obiect `Connection`. Această metodă întoarce un obiect din clasa `Statement`.

```
Statement instructiune = con.createStatement();
```

Putem aplica apoi una dintre metodele `executeQuery()`, `executeUpdate()` sau `execute()` obiectului de tip `Statement` pentru a trimite DBMS-ului instrucțiunile SQL. Metoda `executeQuery()` este folosită în cazul interogărilor care returnează mulțimi rezultat (instanțe ale clasei `ResultSet`), așa cum este cazul instrucțiunilor `SELECT`. Pentru operațiile de actualizare sau ștergere, cum ar fi `INSERT`, `UPDATE` sau `DELETE`, se folosește metoda `executeUpdate()` aplicată obiectului de tip `Statement`, rezultând un întreg care reprezintă numărul înregistrării afectate. Aceeași metodă este folosită pentru interogările SQL DDL, cum ar fi `CREATE TABLE`, `DROP TABLE` și `ALTER TABLE`, în acest caz returnând întotdeauna zero. Metoda `execute()` este folosită în cazul în care se obțin mai mult de o mulțime rezultat sau un număr de linie.


```

ResultSet rs = instructiune.executeQuery("select * from
arhive");
String sql = "insert into arhive values ('Popescu', 'Ion',
'Iasi')";
int raspuns = instructiune.executeUpdate(sql);

```

JDBC 2.0 permite trimiterea spre execuție a mai multor instrucțiuni SQL grupate (eng. *batch*) împreună, această facilitate mărind uneori performanțele. Prezentăm în continuare un astfel de caz:

```

Statement inter = con.createStatement();
con.setAutoCommit(false);

inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");

int [] actualizari = inter.executeBatch();

```

Se observă dezactivarea modului *autocommit*. Acest lucru determină ca modificările rezultate în tabele după execuției metodei `executeBatch()` să nu fie automat salvate permanent (eng. *commit*) sau anulate (eng. *rollback*), aceste operațiuni rămânând la alegerea clientului. Se poate astfel ca în cazul în care una din interogări eșuează, clientul să anuleze efectele tuturor. Pentru alte informații, puteți consulta secțiunea dedicată tranzacțiilor. Metoda `executeBatch()` returnează un tablou în care elementele corespund interogărilor SQL efectuate și reprezintă numărul de linii afectate de instrucțiunile SQL corespunzătoare.

Pentru a executa independent instrucțiunile SQL, se poate păstra modul *autocommit* folosind captarea excepțiilor `BatchUpdateException` aruncate în caz de eroare, așa cum se poate vedea în exemplul următor:

```

try {
inter.addBatch("INSERT INTO arhive VALUES (1, 'Popescu', 'Ioan')");
inter.addBatch("INSERT INTO cantitati VALUES (260, 'file')");
inter.addBatch("INSERT INTO localitati VALUES ('iasi', 'oras')");
int [] actualizari = stmt.executeBatch();
} catch (BatchUpdateException b) {
System.err.println("Actualizari realizate: ");
int [] eroriActualizari = b.getUpdateCounts();
for (int i = 0; i < eroriActualizari.length; i++) {
System.err.print(eroriActualizari[i] + " ");
}
System.err.println("");
}

```

Ștergerea comenzilor dintr-un grup se realizează cu metoda `clearBatch()`.

Un driver JDBC poate să nu ofere suport pentru execuția grupată de instrucțiuni SQL. Pentru a afla dacă se întâmplă sau nu acest lucru, se va folosi metoda `supportsBatchUpdates()` din clasa `DatabaseMetaData`.

Dacă se dorește realizarea de apeluri SQL având date variabile drept intrare, se va folosi clasa `PreparedStatement` care moștenește clasa `Statement`. Prezentăm în continuare modul de construcție a unei astfel de instrucțiuni unde con reprezintă o instanță `Connection`:

```

PreparedStatement instructiune= con.prepareStatement(
"update arhive set nume = ? where prenume like ?");

```

După cum se observă, prin construcție se primește drept intrare o instrucțiune SQL (spre deosebire de cazul `Statement`). În momentul construcției, instrucțiunea SQL primită ca parametru este trimisă direct spre DBMS unde este precompilată. Mai rămâne să dăm valori variabilelor folosind metode `setXX()` corespunzătoare tipurilor de date și să executăm efectiv interogarea după cum se poate vedea în continuare:

```

instructiune.setString(1, "Popescu");
instructiune.setString(2, "Ion");
instructiune.executeUpdate();

```

Această metodă de a face interogări este mai rapidă decât folosirea clasei `Statement` în cazul în care dorim execuția repetată a unei instrucțiuni SQL, deoarece aceasta este compilată o singură dată în momentul creării instanței clasei `PreparedStatement` și folosită apoi repetat, eventual cu valori diferite pentru parametri. Se poate folosi și în cazul în care nu avem parametri, așa cum se poate observa în continuare:

```

PreparedStatement instructiune= con.prepareStatement(
"select * from arhive");
ResultSet rs=instructiune.executeQuery();

```

Din cauza nivelului ridicat de complexitate, procedurile stocate nu le vom prezenta în acest capitol.

Metodele fără parametri de intrare puse la dispoziție de clasa `Connection` pentru cele trei interfețe prezentate în tabelul precedent produc mulțimi rezultat fără cursor deplasabil și nesenzitive la modificări. O mulțime rezultat are cursor pentru poziționare deplasabil, proprietate care poartă numele de *scrolling*, dacă deține un pointer interior care indică înregistrarea accesată la momentul curent, pointer care se poate deplasa programatic. De asemenea, după cum spuneam, mulțimea rezultat nu este senzitivă (eng. *sensitive*) la modificările care pot surveni între timp în tabela interogată. Prin senzitiv se înțelege actualizarea automată a mulțimii rezultat pentru a ilustra dinamic modificările survenite între timp în tabelă.

Driverule care sunt conforme cu versiunea 2.0 a specificației JDBC permit obținerea de mulțimi rezultat senzitive și având cursor deplasabil. Acest lucru este posibil prin specificarea în constructorul instanței `Statement` a uneia dintre constantele prezentate în tabelul care urmează:

Constantă	Explicații
TYPE_FORWARD_ONLY	Acest tip de <code>ResultSet</code> este cel din JDBC 1.0. Mulțimea rezultat nu are cursor deplasabil decât dinspre început spre sfârșit. Modificările făcute în tabelă nu se reflectă în mulțimea rezultat.
TYPE_SCROLL_INSENSITIVE	Mulțime rezultat <i>scrollable</i> , deci cursorul poate fi mutat înainte și înapoi sau pe orice poziție diferită de poziția curentă. De asemenea eventualele modificări făcute între timp în tabelă nu sunt reflectate în <code>ResultSet</code> .
TYPE_SCROLL_SENSITIVE	Mulțime rezultat <i>scrollable</i> . Sensitivitate la modificări.

De asemenea, driverele conforme cu specificația 2.0 a JDBC oferă și posibilitatea actualizării programatice a tabelelor prin intermediul obiectelor `ResultSet`. Tipul de actualizare se specifică prin intermediul uneia dintre constantele prezentate în continuare:

Constantă	Explicații
CONCUR_READ_ONLY	<code>ResultSet</code> -ul nu poate fi modificat programatic. Oferă posibilitatea unui număr nelimitat de conectări concurente la baza de date, deoarece nu se fac modificări asupra tabelii care ar putea genera conflicte. Acest tip de <code>ResultSet</code> este specific driverelor conforme specificației JDBC 1.0.
CONCUR_UPDATABLE	<code>ResultSet</code> -ul și baza de date pot fi modificate programatic. Sunt posibile un număr limitat de conexiuni concurente dat de tipul de concurență ales (a se vedea secțiunea dedicată tranzațiilor). Acest tip de <code>ResultSet</code> este specific driverelor conforme specificației JDBC 2.0.

Prezentăm un exemplu de cod care va determina crearea unei mulțimi rezultat având cursor deplasabil și sensibil la modificări (ordinea parametrilor metodei `createStatement()` este importantă).

```
Statement declar = con.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.
    CONCUR_UPDATABLE);
// la fel se poate crea o instanță a clasei PreparedStatement
ResultSet rez = declar.executeQuery(
    "select nume, prenume from arhive");
```

8.6.4. Procesarea rezultatelor

Pentru parcurgerea simplă a înregistrărilor unui obiect din clasa `ResultSet` folosind JDBC 1.0, putem folosi metoda `next()`, așa cum se poate observa în următoarea secvență de cod.

```
while (rs.next()) // implicit cursor poziționat înainte de prima
```

```
// linie
{
    System.out.println(rs.getString("nume") + ", " +
        rs.getString("prenume") + ", " + rs.getString("oras"));
}
```

În cazul JDBC 2.0, o dată obținută mulțimea rezultat `rez` (a se vedea paragraful precedent) având cursor deplasabil, poate fi folosită pentru a poziționa cursorul în interiorul `ResultSet`-ului. Inițial, cursorul este poziționat înaintea primei linii din mulțimea rezultat. În JDBC 1.0, după cum aminteam, cursorul putea fi mutat doar înainte folosind metoda `next()`. Driverele care sunt conforme cu JDBC 2.0 mai permit poziționarea absolută a cursorului pe orice linie, folosind metoda `absolute(int nr_linie)`, poziționări relative ale cursorului relativ la linia curentă folosind `relative(int nr_salt)`, respectiv deplasări înapoi folosind metoda `previous()`. Dacă argumentul metodei `absolute()` este pozitiv, se va realiza poziționarea pe linia corespunzătoare. Dacă argumentul este negativ, poziționarea se va face prin deplasare de la sfârșit spre început cu un număr de linii egal cu valoarea absolută din argument. Astfel, `absolute(1)` va determina poziționarea pe prima linie, iar `absolute(-1)` va determina întotdeauna poziționarea pe ultima linie. Pentru determinarea liniei curente, se poate folosi metoda `getRow()`. De asemenea, există metodele `first()`, `beforeFirst()` pentru poziționarea pe prima, respectiv înainte de prima linie, `last()` și `afterLast()` pentru poziționarea pe ultima, respectiv după ultima linie. Pentru testarea poziției, avem metodele `isFirst()`, `isLast()`, `isBeforeFirst()`, `isBeforeLast()`. De remarcat că toate poziționările se referă la `ResultSet`, și nu la baza de date interogată.

Prezentăm aceste metode folosite pentru a citi în ordinea inversă liniile din `ResultSet`-ul rezultat în urma interogării precedente:

```
if (rez.isAfterLast() == false) {
    rez.afterLast();
}
while (rez.previous()) {
    String nume = rez.getString(2);
    // nume va detine continutul celulei aflate la intersectia liniei
    // curente din ResultSet cu prima coloana tot din ResultSet
    String prenume = rez.getString("prenume");
    // prenume va detine continutul celulei aflate la intersectia
    // liniei curente din ResultSet cu, coloana prenume din ResultSet
    System.out.println(rez.getRow() + "nume: " + nume + "
        prenume: " + prenume);
}
absolute(-1); // pozitioneaza cursorul pe ultima linie
```

Dacă nu cunoaștem tipul coloanelor `ResultSet`-ului, putem folosi un apel `ResultSet.getMetaData()` pentru a obține o instanță `ResultSetMetaData`, pentru care mai apoi se aplică metoda `getColumnType()`.

În cazul în care driverul folosit este conform JDBC 2.0, clasa `ResultSet` oferă suport pentru actualizări programatice. Prin actualizări programatice vom înțelege actualizări aplicate `ResultSet`-ului, care sunt automat efectuate și asupra bazei de date. În acest fel, baza de date va putea fi modificată fără a efectua instrucțiuni SQL. Metodele pe care clasa `ResultSet` le pun la dispoziție pentru astfel de actualizări sunt `updateXXX()`, unde `XXX` reprezintă un tip de date Java. Spre exemplu, `updateString(String nume_camp, String valoare)`, poate fi folosită pentru a actualiza celula aflată la intersecția între linia curentă și coloana `nume_camp`. Metodele `getXXX` cu care se inspectează `ResultSet`-ul au corespondente `setXXX`.

```
rez.last();
rez.updateInt("nr_curent", 100);
rez.updateString(2, "Popescu");
rez.updateRow();
```

După cum se observă, celula care se vrea actualizată aparține înregistrării curente din `ResultSet`, iar coloana este specificată fie prin indexul său din `ResultSet`, fie prin nume. Prin apelarea metodelor de tip `updateXXX()`, se realizează actualizarea numai în `ResultSet`. Pentru a face actualizarea și în tabla înfășurată, se va folosi mai apoi metoda `updateRow()`. Este foarte important ca apelul acestei metode să se facă atunci când cursorul este poziționat pe înregistrarea care se vrea actualizată. Dacă cursorul este deplasat înainte de apel, driverul va anula actualizarea și, drept consecință, numai mulțimea rezultat va fi modificată, nu și baza de date.

Pentru a evita ca modificările făcute în `ResultSet` să afecteze și baza de date, avem la dispoziție metoda `cancelRowUpdates()`, care, pentru a avea efect, trebuie apelată între metodele `updateXXX` și `updateRow()`. Dacă este apelată după metoda `updateRow()`, nu mai are nici o valoare.

Pe lângă actualizări, API-ul JDBC 2.0 oferă și posibilitatea ștergerii de linii folosind metoda `deleteRow()`.

```
rez.last();
rez.deleteRow();
```

Ștergerea se aplică atât `ResultSet`-ului, cât și tabelii înfășurată. De observat că în exemplul precedent se va șterge ultima linie din `ResultSet`, și nu din tabela înfășurată. Poziția pe care se află în tabelă linia din `ResultSet` ce se vrea ștersă rămâne întotdeauna nedeterminată.

Ultima operație pe care o vom prezenta este inserarea unei linii într-o tabelă. Această operație se realizează prin intermediul unei linii speciale care poartă numele *insert row*. Aceasta este o linie ajutătoare asociată cu o mulțime rezultat, dar nu conținută în aceasta, folosind la completarea cu date a unei linii care va fi mai apoi inserată efectiv în tabelă. Pentru a obține accesul spre *insert row*, clasa `ResultSet` pune la dispoziție metoda `moveToInsertRow()`. Urmează apelul metodelor de tipul `updateXXX` pentru a completa linia *insert row* cu informații. După ce linia a fost completată cu date, se va insera efectiv în tabela curentă printr-un apel `insertRow()`. Mai urmează repoziționarea pe o înregistrare dorită din mulțimea rezultat. Spre

exemplu, pentru a ne repoziționa pe înregistrarea pe care eram înainte de a insera o linie, se va apela metoda `ResultSet.moveToCurrentRow()`.

```
rez.moveToInsertRow();
rez.updateInt(1, 357);
rez.updateString(2, "Popescu");
rez.updateString(3, "Ioan");
rez.insertRow();
rez.first();
```

Apelurile de forma `updateXXX` efectuate când suntem poziționați pe *insert row* nu au nici un efect asupra mulțimii rezultat și nici asupra bazei de date din care a fost generat `ResultSet`-ul. Apelul `insertRow()` va returna o excepție `SQLException` dacă numărul de coloane din `ResultSet` este mai mic decât numărul de coloane din tabela înfășurată. De asemenea, putem obține informații despre datele din *insert row* folosind metode `getXXX`, dar dacă nu am actualizat înainte celula executând un apel de forma `updateXXX`, ceea ce obținem drept rezultat este nedefinit.

Atunci când inserăm o înregistrare în tabelă, JDBC nu oferă nici o modalitate de a afla pe ce poziție a fost introdusă acea înregistrare. Acest lucru este gestionat intern de DBMS.

Pe lângă tipurile de date corespunzătoare specificației SQL-92, prin intermediul unei instanțe `ResultSet` se poate lucra și cu date având tipuri SQL3. Prezentăm în continuare un exemplu de citire a unui tablou dintr-o tabelă:

```
ResultSet rs = stmt.executeQuery("select rezultate from tabela
where nr_curent= 3");
rs.next();
Array tablou = rs.getArray("rezultate");
```

Un exemplu de stocare în mod programatic a unui obiect `Clob`:

```
Clob observatii = rs.getClob("observatii");
PreparedStatement prep = con.prepareStatement(
    "update alta_tabela set comentarii = ? where nr_curent
    < 1000",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
prep.setClob(1, observatii);
```

Dintr-un obiect `Clob` putem să obținem un flux de caractere:

```
java.io.Reader citire = observatii.getCharacterStream();
```

Trebuie remarcat că în cazul tipurilor `CLOB`, `BLOB` și `ARRAY`, prin interogări se obțin referințe spre obiectele de acest tip stocate în bazele de date, în loc ca acestea să fie aduse din DBMS în aplicație, soluția fiind adoptată pentru optimizare. Cu toate acestea, folosirea acestor tipuri, datorită cantității mari de date conținute, determină scăderi în ceea ce privește performanțele. Accesul spre datele de tip `BLOB` și `CLOB`

stocate în baza de date se realizează prin fluxuri de citire scriere. Prezentăm în continuare operația de citire folosind un flux:

```
String sir="";
java.sql.Blob blob=rezultat.getBlob(i);
InputStream inn=blob.getBinaryStream();
InputStreamReader in= new InputStreamReader(inn);
int b;
while (in.read()>-1) {
    b=in.read();
    sir+=b;
}
in.close();
inn.close();
```

În cazul tipurilor definite de utilizator (tipuri distincte și structurate), prezentăm modalitatea de interogare și de obținere a rezultatului:

```
ResultSet rs = stmt.executeQuery(
    "SELECT PUNCT FROM FIGURI WHERE DIM > 3");
while (rs.next()) {
    Struct punct = (Struct)rs.getObject("PUNCT");
    // operatii cu obiectul punct
}
```

În ambele cazuri, și pentru tipuri distincte, și pentru cele structurate, returnarea se face prin valoare, și nu prin referință.

8.6.5. Închiderea unei conexiuni la o bază de date

Atunci când este vorba despre resurse exterioare, așa cum este cazul accesului spre DBMS-uri via JDBC, „colectorul de gunoaie” nu știe nimic despre starea acestor resurse, dacă este sau nu cazul să le elibereze. De aceea, este recomandat ca, după ce procesarea datelor s-a încheiat, programatorii să închidă explicit conexiunile către baza de date. Pentru aceasta se folosește metoda `close()` aplicată obiectului `Connection`. În plus, trebuie închise (înaintea conexiunii) și obiectele `Statement` și `ResultSet` folosind metodele lor `close()`, așa cum se poate vedea în exemplul care urmează.

```
try {
    //...
    rs.close();
    instructiune.close();
}
catch (SQLException e) {
    System.out.println("Eroare la inchidere interogare: " + e.toString());
}
```

```
}
finally {
    try {
        if (conexiuneBazaDate != null) {
            conexiuneBazaDate.close();
        }
    }
    catch (SQLException ex) {
        System.out.println("Eroare la inchidere conexiune: " + ex.toString());
    }
}
```

8.7. Instalarea și configurarea MySQL

8.7.1. Introducere

Unul dintre cele mai simple și cunoscute servere de baze de date este MySQL, care și-a dovedit în timp stabilitatea și fiabilitatea, fiind deseori soluția aleasă pentru gestiunea bazelor de date. Serverul MySQL este oferit gratuit pentru sistemele Unix, Windows și MacOS, sub licența *GNU General Public License (GPL)*, de către firma suedeză MySQL AB. Situl oficial MySQL este <http://www.mysql.com/>, care poate fi consultat pentru a descărca ultimele distribuții MySQL, deja stabile, sau încă în probă, împreună cu documentația aferentă. Tot de aici se poate descărca și driverul JDBC oficial, care poartă numele *Connector/J*, de fapt mai vechiul *mmmysql* preluat sub tutela AB și îmbunătățit. În ceea ce privește documentația, manualul de utilizare MySQL pare a fi referința cea mai importantă pentru dezvoltatori.

După instalare, bazele de date pe care serverul le va deservi clienților vor fi poziționate în directorul `/data` al distribuției. În subdirectorul `/bin` al distribuției MySQL se va găsi executabilul `mysqld`, care permite startarea serverului de baze de date. În același director se află și alte facilități, cum ar fi *WinMySQLAdmin* pentru sistemul de operare Windows, care permite administrarea serverului MySQL dintr-o interfață grafică prietenoasă. Rularea programului *WinMySQLAdmin* se poate face dând click stânga pe butonul *Start*, apoi *Programs* și *Startup*. După selectare, dacă fereastra utilizator se minimizează în bara de stare a ecranului, atunci se face click dreapta și se selectează *Start the server*, urmat de *Show me*. Pentru vizualizarea bazelor de date active se selectează panoul *Databases*, în interiorul căruia se face click dreapta. Apoi se selectează *Create Database* din meniul contextual apărut pentru a crea o nouă bază de date.

O dată baza de date creată, urmează crearea efectivă a tabelelor componente. O metodă va fi prezentată în continuare. Se va crea un fișier text care să cuprindă descrierea unei comenzi SQL `CREATE TABLE`. De exemplu, acesta poate fi intitulat `studenti.sql`, conținând următoarea secvență:

```
CREATE TABLE studenti (
  nr int not NULL auto_increment primary key,
  nume varchar(25) NULL,
  prenume varchar(25) NULL,
  nota int NULL
);
```

Pentru a crea efectiv tabela, se folosește comanda `mysql studenti<studenti.sql` și astfel se creează în directorul `\data`, subdirectorul `\studenti`, cu fișierele `studenti.myi`, `studenti.myd` și `studenti.frm`.

8.7.2. Drivele JDBC pentru MySQL

Datărită faptului că serverul MySQL este *open source*, deci are sursele accesibile și bine documentate, există mai multe implementări de drivele JDBC libere, disponibile pentru MySQL și realizate de diferiți dezvoltatori independenți. O listă a driverelor existente o prezentăm în continuare:

Nume driver	Tip licență	Versiuni JDBC	Pagina de referință	Calea completă
Connector/J	LGPL	1.x și 2.x	http://www.mysql.com/Downloads/Connector-J/mysql-connector-java-2.0.14.zip	com.mysql.jdbc.Driver
Mm(GNU)	LGPL	1.x și 2.x	http://mmmysql.sourceforge.net/	org.gjt.mm.mysql.Driver
Twz	None	1.x	http://www.voicenet.com/~zellert/tjFM/	twz1.jdbc.mysql.jdbcMySQLDriver
Caucho	QPL	2.x	http://www.caucho.com/projects/jdbc-mysql/index.xtp	com.caucho.jdbc.mysql.Driver

Dintre acestea, *mmMySQL* (MM – provine de la Mark Matthews, autorul lui) este driverul recomandat în manualul MySQL, fiind considerat cel mai performant driver gratuit până la apariția lui *Connector/J*, care nu este altceva decât o îmbunătățire a acestuia. Driverul este împachetat într-un fișier `.jar` care trebuie adăugat în `CLASSPATH`. Numele complet al driverului este `org.gjt.mm.mysql.Driver`, sub-protocolul `mysql` și formatul URL-ului JDBC este:

```
jdbc:mysql://[numegazda][:port]/numebazadate[?param1=val1]
[&param2=val2]...
```

De exemplu, URL-ul pentru o bază de date numită *arhiva* și care este pe calculatorul local este `jdbc:mysql://localhost:arhiva`.

O dată cu numele mașinii, al portului și al bazei de date, driverul mai permite setarea unor proprietăți interne cu diverse valori date de utilizatori pe care le prezentăm în tabela următoare:

Nume proprietate	Mod de folosire	Valoare implicită
User	Utilizatorul care se conectează.	nici una
Password	Parola folosită pentru conectare.	nici una
autoReconnect	Va încerca să se reconecteze driverul în cazul în care conexiunea moare? (true/false)	False
maxReconnects	Dacă autoReconnect este setat, de câte ori va încerca driverul să se autoconecteze?	3
initialTimeout	Dacă autoReconnect este setat, timpul inițial de așteptare între reconectări (în secunde).	2
maxRows	Numărul maxim de linii care va fi returnat (0 înseamnă că se vor returna toate liniile)	0
useUnicode	Va folosi driverul setul de caractere Unicode atunci când va mânui șiruri de caractere? (true/false)	False
characterEncoding	Dacă useUnicode este true, ce tip de codare de caractere va folosi driverul?	nici unul

Valorile proprietăților pot fi transmise în momentul realizării conexiunii, folosind diferitele suprascriseri ale metodei `getConnection()` pe care clasa `DriverManager` le pune la dispoziție.

Suprascrisere a metodei	Explicații
public static Connection getConnection (String url)	Permite realizarea unei conexiuni folosind un URL care poate să conțină și valori pentru proprietăți.
public static Connection getConnection (String url, Properties info)	Permite realizarea unei conexiuni folosind un URL și o listă de proprietăți specificate print-un obiect <code>Properties</code> .
public static Connection getConnection (String url, String utilizator, String parola)	Permite realizarea unei conexiuni folosind un URL, un nume de utilizator și o parolă.

Prezentăm în ordine câte un exemplu pentru fiecare tip de conectare:

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/arhiva?user=utilizator&password=parola&
  useUnicode=true&characterEncoding=UTF-8");
```

```
Properties p = new Properties();
p.put("user", "utilizator");
p.put("password", "parola");
p.put("useUnicode", "true");
p.put("characterEncoding", "UTF-8");
Connection con= DriverManager.getConnection(url, p);
```

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/arhiva", "utilizator", "parola");
```

Trebuie remarcat faptul că, prin intermediul proprietăților `useUnicode` și `characterEncoding`, se poate seta ca driverul să folosească codarea Unicode atunci când se lucrează cu șiruri de caractere. Prin această metodă se forțează folosirea diacriticelor. Acest lucru este extrem de important, deoarece, intern, serverul MySQL folosește implicit codarea *Latin1* (dacă nu se setează o alta), și nu Unicode, așa cum este cazul limbajului Java. În fapt, MySQL nici nu oferă suport pentru Unicode până la versiunea 4.0. Atunci când diacriticele sunt trimise spre serverul MySQL după ce au fost culese în componente grafice precum cele Swing, dacă driverul nu este setat pe Unicode, în urma interogărilor vor fi readuse de pe server în mod eronat. Implicit, după cum se observă din tabela anterioară, driverul nu este setat să accepte Unicode. Pentru mai multe informații, puteți consulta capitolul „Internaționalizarea aplicațiilor”.

Una dintre soluțiile cele mai comode pentru gestionarea proprietăților ar fi folosirea fișierelor cu proprietăți, gestionate prin clasa `ResourceBundle` sau prin clasa `Properties`. Presupunem că fișierul `resurse.properties` are drept conținut:

```
Driver=org.gjt.mm.mysql.Driver
URL=jdbc:mysql://localhost/date?user=nume&password=parola
```

Prezentăm în continuare o metodă de a folosi acest fișier cu resurse, pentru a ne conecta la baza de date date și a lista toate intrările din tabela arhive:

```
import java.sql.*;
import java.util.*;

public class Conectare {

    public static void main(String argv[]) {
        Connection con = null;
        ResourceBundle grup = ResourceBundle.getBundle(
            "resurse");
        try {
            String url = grup.getString("URL");
            // inregistram driverul
            Class.forName(grup.getString("Driver"));
            // facem conexiunea
            con = DriverManager.getConnection(url);
            Statement instructiune = con.createStatement();
            // selecteaza toate inregistrarile din tabela arhive
            String sql = "select * from arhive";
            ResultSet rs = instructiune.executeQuery(sql);
            while (rs.next()) {
                System.out.println(rs.getString("nr_curent") + ", " + rs.
                    getString("nume") + ", " + rs.getString("prenume"));
            }
        }
    }
}
```

```
rs.close();
instructiune.close();
    }
    catch( SQLException e ) {
        e.printStackTrace();
    }
    finally {
        if( con != null ) {
            try { con.close(); }
            catch( Exception e ) { }
        }
    }
}
```

Pentru driverul *Connector/J*, tot ceea ce a fost prezentat anterior rămâne valabil, cu observația că clasa care implementează interfața `java.sql.Driver` este acum `com.mysql.jdbc.Driver` și se află într-o arhivă `.jar`, având numele de forma `mysql-connector-j-XXXbin.jar` (unde XXX este versiunea driverului), care trebuie inclus în `CLASSPATH`. Spre exemplu, pentru înregistrarea driverului *Connector/J* se va folosi un apel:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

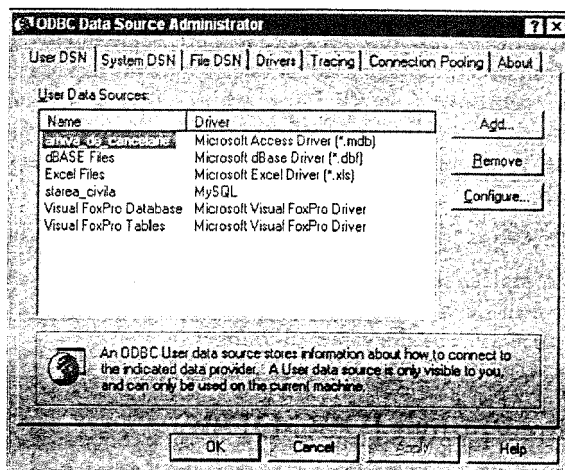
O aplicație completă, care îmbină API-urile JDBC și Swing pentru a realiza conectări la serverul de baze de date MySQL, puteți găsi în secțiunea dedicată componentei grafice `JTable` din capitolul Swing.

8.8. Utilizarea punții JDBC-ODBC

În cazul punții JDBC-ODBC, timpul de acces spre bazele de date este de câteva ori mai mare decât în cazul unui driver de tip 4 și se așteaptă ca această metodă să fie folosită doar experimental, atunci când nu avem încă la dispoziție un alt tip de driver pentru DBMS-ul folosit. Puntea face legătura dintre aplicație și ODBC, care reprezintă un mecanism standard orientat spre programatorii de C/C++, pe care sistemul de operare Windows îl folosește ca interfață universală pentru conectarea cu baze de date de diverse tipuri. Motivele pentru care se folosește o punte, și nu se apelează direct din aplicație API-ul ODBC folosind JNI sunt multiple, cel mai important fiind independența totală față de facilitățile oferite de sistemul de operare peste care este instalată platforma Java. În ODBC, bazele de date sunt reprezentate ca surse de date, identificate printr-un nume (*DNS - Data Source Name*) și accesibile printr-un driver ODBC corespunzător. Putem spune că, în cazul folosirii unui driver de tip 1, avem de a face de fapt cu două drivere și anume puntea propriu-zisă, reprezentată de driverul JDBC-ODBC, respectiv driverul ODBC corespunzător DBMS-ului interogată, care

trebuie instalat în prealabil. Acest lucru poate deseori determina apariția unor erori neașteptate din cauza proastei interacțiuni dintre aceste două drivere.

Prezentăm în continuare un exemplu care ilustrează modul prin care se accesează dintr-o aplicație Java o bază de date creată în Access, sub sistemul de operare Windows 98. Presupunem că baza de date deja a fost creată și are numele *arhiva.mdb*. Pentru a putea face interogări asupra ei, trebuie mai întâi să o introducem ca sursă de date în ODBC. Pentru aceasta, trebuie urmați următorii pași: din ControlPanel alegem ODBC Data Sources (32 bit), ceea ce determină apariția ferestrei ODBC Data Source Administrator. În rubrica User DSN, folosind butonul Add... și scenariul (eng. *wizard*) pe care acesta îl declanșează, selectăm pe rând driverul ODBC (în cazul prezent, spre Access), respectiv baza de date asupra căreia se doresc a se face interogările *arhiva.mdb*, și punem numele acestei surse de date (eng. *Data Source Name*) *arhiva_de_cancelarie*, așa cum se vede în exemplul următor:



Până în acest punct, am creat sursa de date. Pentru conectarea efectivă, trebuie să atașăm aplicației driverul punte spre ODBC. Driverul fiind inclus în distribuția standard, nu mai este necesar să introducem în CLASSPATH calea spre clasele componente, ci este îndeajuns să-l încărcăm dinamic printr-un apel `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")`. Urmează realizarea conexiunii, care se face cel mai convenabil printr-o secvență de forma:

```
// generarea adresei
String url="jdbc:odbc:arhiva_de_cancelarie";
// setarea proprietatilor
java.util.Properties prop = new java.util.Properties();
prop.put("charset", "UTF-8");
prop.put("user", "utilizator");
prop.put("password", "parola");
```

```
// conectarea la baza de date
con = DriverManager.getConnection(url, prop);
```

Prin intermediul proprietății `charset`, se oferă suport pentru internaționalizare, fiind posibilă alegerea unui alt set de caractere, diferit de cel folosit implicit de DBMS. Această facilitate este valabilă numai pentru driverele JDBC-ODBC care vin începând cu distribuția 1.4 a platformei Java.

Prezentăm în continuare o metodă prin care se testează conexiunea spre baza de date *arhiva.mdb*, presupunând că s-a creat mai întâi pentru aceasta sursa de date *arhiva_de_cancelarie*, așa cum am arătat într-un paragraf precedent:

```
static String driver=" sun.jdbc.odbc.JdbcOdbcDriver";
static String parola="";
static String utilizator="";
static String subProtocol="odbc";
static String bazaDate="arhiva_de_cancelarie";
//.....
public static boolean testeazaConexiune()
{
    boolean rezultat=true;
    try {
        Class.forName(driver); // driverul este incarcat dinamic
        String url="jdbc:"+subProtocol+": "+bazaDate;
        System.out.println("A rezultat urmatorul url: "+url);
        //se va afisa: A rezultat urmatorul url: jdbc:
        //odbc:arhiva_de_cancelarie
        Connection conex = DriverManager.getConnection
            (url,utilizator,parola);
        Statement stare = conex.createStatement();
    }
    catch(Exception ex) {
        System.out.println("Eroare la conectare:
            "+ex.toString());
        rezultat=false;
    }
    return rezultat;
}
```

Pentru a testa viteza de acces, putem instala driverul ODBC purtând numele MyODBC (denumit oficial ODBC Connector), care face legătura dintre ODBC și serverul MySQL, driver pe care îl puteți descărca gratuit de la adresa: www.mysql.com/products/myodbc. Acest driver vă permite de asemenea să folosiți un Add-in pentru Access, având numele MyAccess, pe care îl găsiți gratuit la adresa www.mysql.com/portal/software/item-242.html și cu care puteți face conversii ale bazelor de date Access către MySQL.

Codul Java folosit în aplicații rămâne valid după tranziția spre alt sistem de gestiune a bazelor de date, deoarece orice driver de orice tip respectă API-ul JDBC, singurul pas necesar tranziției fiind schimbarea driverului.

8.9. Tranzacții

O tranzacție constă din una sau mai multe instrucțiuni SQL, care sunt executate complet, ca un întreg, iar toate modificările apărute în baza de date accesată sunt salvate permanent folosind metoda `commit()` sau anulate printr-o metodă `rollback()`. De asemenea, orice blocare a datelor conținute în baza de date utilizată de acea tranzacție este anulată prin apelarea uneia dintre metodele anterioare. Când una dintre cele două comenzi `commit()` sau `rollback()` este executată, tranzacția curentă se termină și o alta poate începe.

Implicit, un obiect `Connection` creat este în modul *auto-commit*, ceea ce înseamnă că atunci când o interogare este terminată, metoda `commit()` va fi apelată automat pentru a face permanente modificările apărute. În acest caz se consideră interogarea făcută ca fiind o tranzacție de sine stătătoare. Dacă modul *auto-commit* este dezactivat prin intermediul unei metode `setAutoCommit(false)`, tranziția nou începută nu se va termina decât printr-un apel explicit al uneia dintre metodele `commit()` sau `rollback()` și va îngloba toate instrucțiunile SQL executate între timp. Operația de salvare permanentă sau de anulare are efect asupra tuturor interogărilor din tranzacție. Dacă una dintre instrucțiunile SQL nu se execută cu succes, se va executa explicit `rollback()`, ceea ce înseamnă că toate modificările făcute de acele interogări sunt anulate, ajungându-se la starea de dinaintea începerii tranzacției. Altfel, toate modificările determinate de instrucțiunile SQL care compun tranzacția sunt comise printr-un apel explicit `commit()`. Acest lucru se întâmplă din cauza proprietății de atomicitate a tranzacțiilor (a se vedea subsecțiunile următoare).

După cum se observă, începerea unei tranzacții nu necesită nici un apel explicit, fiind implicit inițiată în momentul terminării unei alte tranzacții sau prin apelul `setAutoCommit(false)`.

Orice driver JDBC trebuie să suporte tranzacțiile. În fapt, pentru ca un driver să poată fi acceptat ca fiind compatibil JDBC, trebuie să suporte tranzacțiile.

8.9.1. Motivații pentru folosirea tranzacțiilor

Câteva motive pentru care folosirea tranzacțiilor își găsește justificare ar putea fi:

Necesitatea de a realiza operații atomice – presupune situația în care se dorește ca mai multe operații separate să se execute împreună, într-o manieră atomică. Această abordare este utilă, spre exemplu, în cazul în care dorim să facem modificări asupra mai multor tabele aflate eventual în baze de date diferite pe mașina locală sau în rețea, modificări care își au locul doar făcute împreună. Mai particular, putem da drept exemplu cazul transferului de bani dintr-un cont în altul, transfer care presupune ca, o dată suma de bani citită și modificată din contul sursă, să fie scrisă și în contul

destinație. Dacă operația nu decurge în mod atomic, se va întâmpla, spre exemplu, ca suma să nu fie transferată efectiv și să se regăsească și în sursă, și în destinație, ceea ce reprezintă un transfer incorrect (inconsistență). Tranzacțiile rezolvă această problemă prin garantarea principiului „*totul sau nimic*”.

Căderi ale mașinii sau ale rețelei – în cazul în care bazele de date sunt distribuite în rețea, situațiile ce pot determina excepții cresc. De asemenea, mașina pe care se află bazele de date poate să cadă la un moment dat. La fel, serverul de baze de date. În cazul în care căderea se petrece atunci când se face o scriere în baza de date, aceasta poate să fie detriorată irecuperabil. Tranzacțiile sunt cele care oferă suport pentru recuperarea datelor.

Partajarea acelorași date de mai mulți utilizatori – într-un intranet apare frecvent posibilitatea accesării unei aceleași baze de date de mai mulți clienți în același timp. Conexiunile concurente pot determina o serie de erori, printre care amintim:

Citiri murdare (eng. dirty reads) – această problemă apare în cazul în care un client citește date dintr-o bază de date care nu a fost salvată definitiv pe suportul de stocare de un alt client prin operația `commit()`. Drept exemplu, presupunem următoarea succesiune de pași:

1. Clientul C1 citește întregul X din baza de date. Inițial presupunem că X este 0.
2. Clientul C1 adună 10 la X și astfel baza de date are acum X=10. Presupunem că C1 nu a aplicat încă `commit()`, ceea ce ar fi determinat salvarea definitivă a bazei de date pe suport.
3. Un alt client C2 citește întregul X din baza de date. Valoarea pe care o citește din X este 10.
4. Clientul C1 renunță (eng. *abort*) la tranzacție, ceea ce determină ca X din tabelă să devină 0.
5. Clientul C2 adună 20 la X și îl salvează în baza de date. Acum baza de date va conține X=30.

După cum se poate observa, clientul C2 folosește la pasul 5., o valoare a lui X care nu mai este valabilă din cauza pasului 4. Deci o aplicație a folosit o dată tabelă înainte ca ea să fie stocată permanent, ceea ce determină o eroare în logica aplicației.

Citiri nerepetabile (eng. unrepeatable reads) – această problemă apare atunci când un client citește date dintr-o tabelă, dar între timp acestea sunt modificate de un alt client. Ca succesiune de pași:

1. Clientul C1 citește întregul X din baza de date.
2. Clientul C2 suprascrisă întregul din tabelă cu o nouă valoare.
3. Clientul C1 citește iarăși întregul X. Surprinzător, acesta nu mai are aceeași valoare.

Problema fantomelor (eng. phantoms) – această problemă presupune apariția unei noi mulțimi de date între două operații de citire făcute de un același client. Pe pași, problema poate fi privită astfel:

1. Clientul C1 interoghează aplicația și obține o mulțime de date rezultat.

2. Clientul C2 inserează noi date în baza de date.
3. Clientul C1 reapelează aceeași interogare, dar, surprinzător, rezultatul obținut este altul.

Diferența între citiri nerepetabile și problema fantomelor este aceea că în primul caz este vorba de modificări de date existente, iar în al doilea caz este vorba de date cum ar fi înregistrări sau coloane noi, care nu existau anterior.

8.9.2. ACID

În cazul folosirii tranzacțiilor, un număr de patru principii sunt totdeauna garantate pentru operațiile efectuate. Acestea poartă și numele de proprietăți ACID ale tranzacțiilor, denumire care vine de la abrevierea celor patru nume de proprietăți: Atomicitate (eng. *Atomicity*), Consistență (eng. *Consistency*), Izolare (eng. *Isolation*) și Durabilitate (eng. *Durability*).

Prezentăm în continuare descrierea celor patru proprietăți:

Atomicitate – garantează că mai multe operații sunt grupate împreună și apar ca o singură unitate continuă de lucru. De asemenea, atomicitatea garantează că mai multe operații sunt executate toate odată sau nici una.

Consistență – garantează că starea sistemului rămâne consistentă după ce o tranzacție este terminată. Prin stare consistentă se înțelege o stare în care sistemul respectă o anumită regulă sau, mai general, anumite reguli. Sistemul poate trece și prin stări temporare de inconsistență. Dar, în final, tranzacțiile garantează consistența sistemului.

Izolare – permite ca mai mulți clienți să citească și să scrie într-o bază de date fără să știe unul de altul, deoarece tranzacțiile sunt izolate una de alta. Este metoda prin care se rezolvă problemele legate de partajarea resurselor (a se vedea subsecțiunile anterioare). Acest lucru este util pentru cazul mai multor clienți care accesează baza de date în același timp, fiecare având impresia că el este singurul conectat. Tranzacțiile asigură izolarea prin intermediul sincronizărilor la nivelul inferior al DBMS, prin blocări (eng. *locks*) automate ale datelor sensibile, care asigură că o dată ce un client accesează zona sensibilă (critică), nimeni altcineva nu mai poate face acest lucru. Încă o dată trebuie remarcat faptul că, prin folosirea tranzacțiilor, programatorii nu mai trebuie să scrie cod pentru tratarea concurențelor multiple, deoarece de acest lucru se ocupă DBMS-ul. Tot ce trebuie să facem este să specificăm tipul de acces.

Durabilitate – garantează că toate actualizările permanent salvate folosind `commit()` supraviețuiesc căderilor de sistem, erorilor de rețea, căderii hard disk-ului sau căderilor de tensiune. Acest lucru este posibil datorită faptului că tranzacțiile păstrează un fișier de siguranță în care sunt înregistrate toate operațiile asupra bazelor de date, înainte ca acestea să fie efectiv comise. Datele salvate permanent pot fi astfel reconstituite prin reaplicarea pașilor salvați în acest fișier.

Vom vorbi mai multe în continuare despre izolarea tranzacțiilor, care necesită o abordare specială, din pricina problemelor ce pot apărea în momentul partajării aceluiași resurse de către mai mulți clienți.

8.9.3. Niveluri de izolare

Dacă un DBMS suportă tranzacții concurente, programatorul aplicației care se conectează la acel DBMS poate specifica nivelul de izolare al tranzacțiilor efectuate. Izolarea poate fi strictă sau relaxată. O izolare strictă determină separarea totală a tranzacțiilor, care se realizează prin restricții impuse asupra datelor din tabelele bazei, soldate cu o scădere a performanțelor. De aceea trebuie ales cu grijă nivelul de izolare de care este nevoie. Există mai multe niveluri de izolare, descrise prin cinci constante din clasa `Connection`, pe care le vom prezenta în tabelul care urmează:

Nivel izolare	Explicații
<code>TRANSACTION_NONE</code>	Nu sunt permise tranzacțiile.
<code>TRANSACTION_READ_UNCOMMITTED</code>	Nu oferă nici un fel de izolare, dar garantează performanțe înalte în ceea ce privește viteza de acces. Acest nivel de izolare nu este indicat a se folosi în sisteme cu conexiuni concurente și, mai ales, în cele cu date sensibile, cum ar fi sistemul bancar, din cauza problemelor inevitabile care vor apărea.
<code>TRANSACTION_READ_COMMITTED</code>	Rezolvă problema regiunilor murdare. Folosind acest nivel de izolare, nu se vor citi datele care au fost modificate în tabela accesată, fără a fi salvate definitiv prin metoda <code>commit()</code> . <code>Read committed</code> este nivelul implicit de izolare pentru cele mai multe servere printre care Oracle și MySQL.
<code>TRANSACTION_REPEATABLE_READ</code>	Rezolvă problemele anterioare și problema citirii nerepetabile.
<code>TRANSACTION_SERIALIZABLE</code>	Rezolvă problemele anterioare și problema fantomei. Folosind acest nivel, tranzacțiile sunt izolate total, ceea ce înseamnă că ele se vor executa secvențial, independent una față de alta. Alegerea acestui nivel va determina încetinirea simțitoare a tranzacțiilor concurente.

Atunci când un obiect `Connection` este creat, nivelul de izolare al conexiunii depinde de driver. Programatorii pot face apeluri de forma `setTransactionIsolation(TRANSACTION_REPEATABLE_READ)` și un nou nivel de izolare va fi valabil pe tot parcursul conexiunii. Schimbarea modului de izolare a tranzacțiilor în timp ce acestea se desfășoară nu este recomandat, deoarece determină execuția imediată a metodei `commit()`, care va face permanente modificările din tabelă.

8.10. Concluzii

Acest capitol prezintă aspectele fundamentale legate de conectarea aplicațiilor Java la sistemele de gestiune a bazelor de date (DBMS). Unele principii, precum sursele de date și procedurile stocate, nu fac subiectul acestui capitol.

Pentru început au fost descrise aspecte legate de API-ul JDBC și specificațiile care stau la baza acestui API. Urmează clasificarea driverelor JDBC și adresele paginilor Web de unde se pot afla mai multe informații și de unde, eventual, acestea se pot descărca. Mai apoi, este pus la dispoziția cititorilor un scurt rezumat al interogărilor SQL, dintre care unele sunt folosite în exemplele capitolului. Urmează maparea dintre tipurile SQL și tipurile Java, respectiv informații despre tipurile SQL3 suportate de JDBC 2.0.

Accesarea unei baze de date este prezentată pe larg, urmată de prezentarea DBMS-ului MySQL, cu două dintre driverele cele mai performante, și a punții JDBC-ODBC. În final se va face o incursiune în tranzacții și sunt prezentate în special o serie de probleme determinate de conexiuni multiple, precum și metodele de rezolvare a lor.

Ca o concluzie, putem spune că JDBC reprezintă unul dintre cele mai mature API-uri independente pentru accesarea bazelor de date și că se bucură de sprijinul unor giganti informatici (precum ORACLE), care și-au fundamentat produsele lor pe tehnologii Java.

8.11. Test grilă

Întrebarea 8.11.1. Ce reprezintă un sistem de gestiune al bazelor de date (DBMS)?

- O aplicație care accesează baze de date ținute de un server.
- Un sistem de reguli și convenții care reglementează schimbul de date între componenții unei arhitecturi client-server.
- Un sistem care permite gestionarea bazelor de date și oferă modalități ca acestea să fie accesate de clienți via conexiuni.
- Un gestionar de conexiuni între un client și un server de baze de date.

Întrebarea 8.11.2. Ce reprezintă un driver JDBC?

- O componentă fizică ce trebuie instalată ca un modul al calculatorului, pentru a ne putea conecta la un server de baze de date.
- O componentă soft care intermediază între un client Java și un DBMS (SGBD).
- O aplicație server care trimite date clienților în urma unor interogări SQL.

Întrebarea 8.11.3. Ce reglementează specificația JDBC?

- Cum trebuie construiți clienții care folosesc driverele JDBC.
- O serie de interfețe standard ce trebuie suprascrise de creatorii de drivere JDBC.
- Sintaxa SQL.

- Versiunile driverelor JDBC.
- Concordanța dintre JDBC și ODBC.

Întrebarea 8.11.4. Fie următorul apel:

```
Statement declar = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.
    CONCUR_UPDATABLE);
```

Ce proprietăți vor avea instanțele ResultSet obținute folosind acest obiect Statement?

- Vor avea cursor deplasabil numai înainte și vor fi senzitive la modificările din tabelele interogate.
- Vor avea cursor poziționabil pe orice linie, permițând modificarea programatică a tabelului interogate.
- Vor fi insenzitive la modificările survenite în tabele, permițând modificarea programatică a tabelului interogate.

Întrebarea 8.11.5. Fie tabela t cu 10 înregistrări și având coloanele a și b, respectiv următoarea secvență de cod:

```
ResultSet rez = instructiune.executeQuery("select * from t");
rez.last();
rez.updateInt("a", 100);
rez.updateString(2, "I");
rez.first();
```

Care dintre afirmațiile de mai jos sunt adevărate după execuția codului?

- Se va adăuga în tabela t o nouă înregistrare.
- Se va modifica ultima linie din tabel pentru a actualiza elementul de pe coloana a la 100 și elementul de pe coloana b la "I".
- Tabela va rămâne nemodificată.

Întrebarea 8.11.6. Ce reprezintă un obiect java.sql.Clob obținut în urma unui apel getClob al clasei ResultSet?

- Un șir de caractere cu o lungime mai mare decât 256.
- O referință spre un obiect de tip SQL BLOB stocat de DBMS.
- Un tip de date SQL definit de utilizator.

Întrebarea 8.11.7. Fie secvența de cod:

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost/arhiva?user=utilizator&password=parola&
    useUnicode=true&characterEncoding=UTF-8");
```

Care dintre afirmațiile următoare este adevărată?

- a) Codul permite obținerea unei conexiuni la serverul MySQL folosind driverul MMySQL.
- b) Codul permite realizarea unei conexiuni la serverul MySQL și cere driverului să folosească standardul Unicode în comunicație.
- c) Codul înregistrează driverul Connector/J și realizează o conexiune la serverul MySQL folosind acest driver.

Întrebarea 8.11.8. Ce reprezintă ODBC?

- a) Implementarea obiectuală a driverelor JDBC.
- b) Un driver JDBC care permite conectarea la baze de date Access.
- c) O sursă de date.
- d) Un server DBMS.

Întrebarea 8.11.9. Ce determină un apel `commit()`?

- a) Terminarea unei tranzacții.
- b) Începerea unei noi tranzacții.
- c) Anularea unei tranzacții.

Întrebarea 8.11.10. Care dintre afirmațiile următoare reprezintă problema citirii murdare (eng. *dirty read*)?

- a) Un client citește date dintr-o tabelă, dar între timp alt client modifică respectivele date. Recitind, primul client obține alte informații.
- b) Un client scrie o informație în baza de date fără a salva definitiv datele și un alt client folosește între timp informația. Primul client renunță, iar al doilea modifică valoarea citită și o salvează.
- c) Un client aplică metoda `commit`, în timp ce altul aplică metoda `rollback`.

8.12. Exerciții propuse spre implementare

Exercițiul 8.12.1. Să se realizeze o aplicație care creează tipuri SQL utilizator, înfășurând clase Java deja existente, permițând astfel stocarea obiectelor instanțe ale respectivelor claselor Java în baze de date gestionate de un DBMS oarecare.

Exercițiul 8.12.2. Să se implementeze o aplicație care oferă toate proprietățile și informațiile despre un DBMS la care aceasta poate crea o conexiune folosind un driver JDBC. Se va folosi clasa `DatabaseMetaData`.

Exercițiul 8.12.3. Scrieți o clasă Java care testează posibilitatea realizării de conexiuni spre diverse DBMS-uri, folosind diverse drivere.

Exercițiul 8.12.4. Să se creeze o aplicație care face conversia bazelor de date dinspre un DBMS spre un altul. În particular, să se implementeze posibilitatea de a face conversii de baze de date dinspre MySQL spre Access și viceversa.

Exercițiul 8.12.5. Testați diferențele în viteza de execuție pentru interogările executate asupra serverului de baze de date MySQL, folosind toate driverele gratuite existente.

Exercițiul 8.12.6. Testați pe DBMS-ul preferat problemele care pot apărea din cauza neizolării tranzacțiilor. Aplicați nivelurile de izolare pentru a observa noii timpi de acces spre DBMS.

Exercițiul 8.12.7. Realizați o metodă de salvare a interogărilor SQL efectuate asupra unui DBMS într-un fișier de siguranță XML, înainte ca acestea să fie trimise spre server și executate. Realizați o modalitate de apelare pas cu pas a interogărilor salvate în acel fișier, pentru a recupera o tabelă dacă aceasta a fost distrusă întâmplător.

8.13. Proiecte propuse spre implementare

Proiectul 8.13.1. Realizați o aplicație Java care să administreze bazele de date ținute de un DBMS. Aplicația va arăta bazele de date, va permite crearea de noi baze de date și tabele, actualizări asupra tabelor componente etc. DBMS-ul va putea fi ales în timpul rulării aplicației administrator, iar driverul va putea fi descărcat de pe Internet și încărcat în aplicație în timpul rulării acesteia.

Proiectul 8.13.2. Creați o aplicație Java care să salveze obiectele Java în tabele prin maparea câmpurilor din obiecte la câmpurile tabelor. Perfecționați acest procedeu pentru a reprezenta o alternativă la serializarea obiectelor Java.

Proiectul 8.13.3. Creați o aplicație Java care să salveze în fișiere XML datele ținute de un DBMS. Aceasta va reprezenta o modalitate de a realiza copii de siguranță ale tabelor. Să se asigure o metodă de restaurare a tabelor a căror salvare s-a realizat.

Proiectul 8.13.4. Creați un adaptor universal care să permită vizualizarea într-un `JTable` a unui `ResultSet` rezultat în urma unei interogări SQL. Vor varia numele bazei de date, driverul și interogarea SQL. A se vedea exemplul prezentat în capitolul Swing la secțiunea dedicată componentei grafice `JTable`. Tabela va putea fi actualizată direct din interfața grafică. Se vor folosi și avantajele oferite de drivere conforme specificației 2.0 sau 3.0.

Proiectul 8.13.5. Realizați o aplicație care să testeze viteza de accesare a datelor gestionate de mai multe DBMS-uri. Aplicația se va putea conecta la mai multe baze de date, aflate eventual pe mai multe DBMS-uri, și va permite realizarea de comparații între diverșii timpi de accesare obținuți.

Proiectul 8.13.6. Realizați o aplicație pentru controlul versiunilor (CVS) care să țină atât surse Java ca tipuri `CLOB`, cât și clase ca tipuri `BLOB` în baze de date și să le furnizeze la cerere clienților. În cazul în care baza de date curent interogată nu răspunde, se va oferi posibilitatea conectării la o alta, eventual gestionată de un alt DBMS.

Proiectul 8.13.7. Realizați o aplicație (sau un applet) Java care să permită celor care o accesează să rezolve teste grilă și să primească punctajul obținut. Testarea se va face contra timp, cu posibilitatea creșterii dificultății la cererea clientului. Testele vor fi împărțite pe domenii și vor fi stocate în baze de date.

9. Programarea rețelelor

9.1. Cuvinte cheie

- canal de comunicație (eng. *socket*), datagramă, client/server
- TCP/IP, UDP/IP
- adrese IP, URL
- apelul metodelor la distanță (eng. *Remote Method Invocation*)
- programare distribuită

9.2. Introducere

Programarea rețelelor este de obicei descrisă folosind conceptele de *client* și *server*. Un *server* este o aplicație care rulează pe un calculator gazdă (*host*) care furnizează o conexiune și informații utile din momentul stabilirii acelei conexiuni. Uneori, din abuz de limbaj, se utilizează termenul de server atât pentru aplicația care rulează, cât și pentru calculatorul gazdă pe care rulează aplicația. Un *client* este o aplicație care rulează pe un calculator din rețea, căutând să stabilească o conexiune cu un server. De regulă, se pot conecta mai mulți clienți simultan la un server.

Serverul este un furnizor de informații, iar clientul este un consumator. Clientul are nevoie de informații, se conectează la un server căruia îi adresează o cerere. Respectivul server trimite înapoi un răspuns care poate conține informația cerută, fie indică faptul că nu posedă respectiva informație. Clientul poate să continue emiterea de cereri, eventual altor servere.

Pentru Web navigatoarele sunt clienți, iar serverele Web sunt servere. De exemplu, dacă indicăm navigatorului adresa `http://www.infoiasi.ro/index.html`, atunci acesta va adresa o cerere serverului `www.infoiasi.ro` pentru documentul `index.html`. Dacă serverul nu posedă respectiva pagină, atunci va returna un cod de eroare. În caz contrar, va trimite pagina Web cerută. Pentru vizualizarea documentului Web, navigatorul poate avea nevoie și de alte documente. În acest caz, va trebui să mai adreseze câteva cereri serverului Web. Eventual, dacă pagina conține o imagine aflată pe alt server, navigatorul va adresa o cerere și respectivului server.

Pentru a se putea realiza o comunicare între clienți și server, aceștia vor trebui să respecte un set de reguli (*protocol*). De exemplu, mai întâi trimite clientul cererea, după care i se trimite un răspuns. Mesajele trimise trebuie să fie înțelese de cei doi parteneri.

9.2.1. Adrese, porturi și socketuri

Pentru a comunica între ele, calculatoarele trebuie să fie conectate fizic. Cea mai mare rețea cunoscută este Internetul, care este de fapt o rețea de rețele de calculatoare.

Un calculator este unic identificat într-o rețea prin adresa de IP (eng. *Internet Protocol*). Aceasta este formată din patru numere cuprinse între 0 și 255, separate prin simbolul punct (de exemplu: 193.231.30.255). Identificarea unui calculator din rețea se mai poate realiza prin numele asociat. De exemplu, `thor.info.uaic.ro` este numele calculatorului cu adresă de IP mai sus menționată. Aceste nume se mai numesc și *domenii*. Domeniile sunt mult mai ușor de reținut.

Numele de domeniu denotă o mașină specifică unui număr de niveluri, niveluri ce se precizează de la general la mai specific în ordine de la dreapta la stânga. De exemplu, adresa `thor.info.uaic.ro` semnifică mașina `thor`, din domeniul Facultății de Informatică, din cadrul Universității „Al.I. Cuza”, România.

Pentru a schimba date, clienții și serverele au nevoie de mai mult decât adrese. De exemplu, multe servere pot rula simultan pe aceeași mașină, deci au aceeași adresă IP.

Într-un calculator, mecanismul utilizat pentru stabilirea unei „întâlniri” se numește *port*. Porturile sunt numere întregi cuprinse între 0 și 65535. De regulă, valorile mai mici de 1024 sunt rezervate pentru servicii predefinite (de exemplu: poșta electronică, *ftp* – transfer de fișiere, *http* – transfer de documente hipertext). Programatorii pot utiliza pentru propriile aplicații porturile mai mari de 1024.

Combinatia dintre adresa IP și numărul de port este întrebuințată pentru crearea unui termen abstract, numit *socket*. Acesta se mai numește în literatura de specialitate *canal de comunicație*, sau simplu, *conexiune*. Un client caută serverul, stabilește o conexiune prin care informația poate fi trimisă, iar la final se va deconecta lăsând liber socketul pentru o utilizare ulterioară. Un socket furnizează facilități pentru crearea de fluxuri de intrare și ieșire, care permit datelor să fie schimbate între client și server.

Pentru un socket, numărul de port poate fi specificat prin program sau asignat de către sistemul de operare. Când un socket trimite un pachet, acestuia trebuie să-i adăugăm informații despre destinația pachetului, cum ar fi:

- adresa mașinii din rețea care trebuie să primească pachetul;
- port pentru indicarea socket-ului corespunzător care va primi datele.

Atunci când se stabilește o conexiune, atât clientul, cât și serverul, vor avea câte un socket. Comunicarea efectivă se va realiza între socket-uri.

În mod necesar, socketul serverului va indica la crearea numărul portului, după care va aștepta realizarea de conexiuni cu clienții. Numerele de port pentru socketurile serverului sunt numere cunoscute pentru programele client. De exemplu, un server FTP (File Transfer Protocol) utilizează un socket care „ascultă” la portul 21. Dacă un program client dorește să comunice cu un server FTP, acesta va ști să contacteze socketul care ascultă la portul 21.

De obicei, sistemul de operare stabilește numărul portului pentru socketurile client. Alegerea portului pentru client nu prezintă importanță, întrucât serverul nu trebuie să știe dinainte de pe care port vin clienții. Când un socket client trimite un pachet către un socket server, pachetul conține și numărul de port și adresa mașinii clientului. Astfel, cu ajutorul informațiilor primite, serverul este capabil să știe cine a adresat cererea. Aceste date sunt utilizate pentru a răspunde clientului.

Când se folosesc socketuri, trebuie să decidem care tip de protocol dorim să-l utilizăm pentru transportul datelor în rețea:

- protocol orientat pe conexiune;
- protocol neorientat pe conexiune.

În cazul protocolului orientat pe conexiune, un socket client stabilește o conexiune cu un socket server când acesta este creat. O dată stabilită conexiunea, un protocol orientat pe conexiune asigură că datele s-au trimis sigur, adică:

- fiecare pachet trimis este și primit. De fiecare dată când un socket trimite un pachet, acesta așteaptă o confirmare că pachetul s-a primit cu succes. Dacă socketul nu primește confirmarea într-un timp precizat, socketul trimite din nou pachetul. Socketul continuă să trimită pachete până când transmisia este cu succes sau când decide că transmisia este imposibilă;

- pachetele sunt citite de socketul destinație în aceeași ordine în care au fost trimise. Datorită modului în care funcționează rețelele, pachetele pot ajunge într-o ordine diferită față de cea în care au fost trimise. Protocolul orientat pe conexiune va permite socketului destinație să restabilească ordinea în care au fost trimise pachetele. În comunicarea orientată conexiune este necesară participarea simultană a celor două aplicații.

Un protocol neorientat pe conexiune permite cea mai rapidă trimitere a pachetelor. Acesta nu garantează că pachetele care sunt trimise sunt citite în aceeași ordine de programul destinatar. Mai mult, programul destinatar nu trebuie neapărat să fie disponibil pentru recepționarea datelor. Expeditorul trimite informațiile, iar destinatarul le va citi eventual mai târziu, după care va răspunde. În schimb, este garantată primirea pachetelor (acestea nu vor fi pierdute). Iată două situații când protocoalele neorientate pe conexiune sunt frecvent preferate celor orientate pe conexiune:

- Când trebuie trimis doar un singur pachet și garantarea trimiterii nu este crucială, atunci un protocol neorientat pe conexiune elimină timpul pierdut pentru crearea și distrugerea unei conexiuni. De exemplu, protocolul orientat pe conexiune TCP/IP (*Transmission Control Protocol/Internet Protocol*) utilizează șapte pachete pentru trimiterea unui singur pachet, în timp ce protocolul neorientat pe conexiune UDP/IP (*User Datagram Protocol/Internet Protocol*) folosește doar unul. Un protocol pentru obținerea timpului curent, de regulă, recurge la un protocol neorientat pe conexiune pentru a minimiza întârzierile;
- Pentru aplicații în timp real, cum ar fi aplicațiile audio/video, garantarea unei transmisii sigure nu este un avantaj. Așteptările unor date care întârzie pot implica pauze importante în aplicația respectivă. Tehnicile de trimitere audio/video care utilizează un protocol neorientat pe conexiune au fost dezvoltate și funcționează mai bine. De exemplu, aplicația *RealAudio* utilizează un protocol neorientat pe conexiune pentru transmiterea sunetelor prin rețea. De asemenea, protocolul RTP (*Real Time Protocol*) este tot neorientat conexiune.

Diferențele dintre conexiunile TCP și UDP pot fi descrise prin următoarea analogie. TCP este asemeni unui apel telefonic: numărul de telefon este format o dată, apoi se stabilește o conexiune care este valabilă până când este întreruptă legătura de unul din participanți. Pe de altă parte, UDP este echivalent cu sistemul de poștă obișnuită. Fiecare scrisoare (mesaj) este pusă într-un plic (pachet) separat și vor ajunge separat sau nu la destinație. Pachetele pot fi pierdute și nu este sigur dacă acestea vor fi recepționate în ordinea în care au fost trimise. Cu toate acestea, dacă un pachet ajunge, suntem siguri că acesta este intact! Ambele protocoale TCP și UDP transmit datele prin intermediul IP-ului, împărțind mesajul în pachete IP individuale.

9.3. Programarea rețelor prin intermediul conexiunilor

Pachetul `java.net` furnizează două mecanisme de bază pentru accesarea datelor și a resurselor prin intermediul unei rețele:

- *socketul*, care este mecanismul fundamental. Un socket permite programelor să schimbe grupuri de octeți (pachete). Există clase în pachetul `java.net` care oferă suport pentru lucrul cu socketuri: Clasele `Socket` și `ServerSocket` sunt utile pentru realizarea unui transfer orientat conexiune, iar `DatagramSocket`, `DatagramPacket` și `MulticastSocket` pentru cele neorientate conexiune.
- *URL* (*Uniform Resource Locator*), care este un mecanismul de nivel înalt. Pachetul `java.net` conține clasa `URL` pentru accesarea și obținerea unei resurse dintr-o rețea.

Implementarea socketurilor în Java este bazată pe biblioteca `Socket` care a fost parte originală din BSD UNIX. De aceea, există foarte multe similarități cu socketurile din alte limbaje de programare, indiferent de platforma acestora.

Un socket TCP pentru client corespunde unei instanțe a clasei `Socket`, iar pentru un server, unei clase `ServerSocket`. O dată stabilită conexiunea, se pot trimite sau primi fluxuri de la aplicația de la distanță, comunicarea realizându-se prin intermediul fluxurilor. Se pot utiliza fluxurile de octeți de tip `Filter` pentru adăugarea funcțiilor de nivel înalt pentru canalul de comunicare.

9.3.1. Clasa `ServerSocket`

Pentru a implementa un server orientat conexiune, trebuie să urmărim următorii pași:

- se creează o instanță a clasei `ServerSocket` pentru a crea un socket care va aștepta la un port precizat pentru stabilirea conexiunilor;
- când se va accepta o conexiune (prin apelul metodei `accept()`), se va crea un obiect `Socket` care încapsulează conexiunea;
- se va utiliza obiectul `Socket` creat anterior pentru a obține un flux de intrare de tip `InputStream`, respectiv unul de ieșire `OutputStream`, în vederea citirii și scrierii informațiilor către, respectiv de la client. Comunicarea cu clientul se va realiza deci prin intermediul fluxurilor;
- se vor putea, opțional, crea noi fire de execuție pentru fiecare conexiune nouă, astfel încât să se poată accepta în continuare noi conexiuni în timp ce celelalte cereri sunt servite.

Pentru construirea obiectelor `ServerSocket` trebuie să alegem un port pe mașina locală cuprins între 1 și 65535. Dacă se selectează numărul de port 0, atunci sistemul de operare va atribui un port liber și valid de fiecare dată când rulează aplicația. Numărul de port ales trebuie apoi comunicat clienților pentru a se putea conecta la server.

După ce a fost creat un obiect `ServerSocket` se vor putea accepta conexiuni cu clienții. Dacă apar mai multe cereri de conexiune din partea clienților, acestea vor fi depuse într-o coadă de așteptare și eliminate una câte una de îndată ce serverul apelează metoda `accept()`. Prin intermediul constructorului clasei `ServerSocket` se permite specificarea numărului de conexiuni care pot fi în coada de așteptare; sistemul de operare va respinge orice cerere de conexiune ulterioară care apare în cazul în care coada este plină.

Constructorii clasei `ServerSocket` sunt:

Prototipul constructorului	Descriere
<code>ServerSocket()</code>	Creează un socket pentru server, nelegat la un port.
<code>ServerSocket(int port)</code>	Creează un socket pentru server, legat la portul indicat. Coada de așteptare poate conține cel mult 50 de cereri pentru acceptarea conexiunii.
<code>ServerSocket(int port, int dimCoadă)</code>	În plus față de constructorul precedent se stabilește și dimensiunea cozii de așteptare.
<code>ServerSocket(int port, int dimCoadă, InetAddress bindAddr)</code>	Al treilea parametru este obiect al clasei <code>InetAddress</code> și indică adresa IP a mașinii pentru care se acceptă conexiuni. Dacă se transmite <code>null</code> , se vor accepta conexiuni pentru orice adresă a mașinii.

Toți constructorii acestei clase pot arunca excepția `IOException`, iar în afara de primul și excepția `SecurityException`.

Cele mai importante metode publice ale clasei `ServerSocket` sunt cele din tabelul de mai jos:

Prototipul metodei	Descrierea metodei
<code>Socket accept()</code> throws <code>IOException</code>	Blochează execuția firului de execuție curent până când un client realizează o cerere pentru conexiune. Este returnat obiectul de tip <code>Socket</code> corespunzător conexiunii.
<code>void close()</code> throws <code>IOException</code>	Închide socketul. Orice metodă <code>accept()</code> blocată a unui fir de execuție va arunca excepția <code>SocketException</code> .
<code>InetAddress getInetAddress()</code>	Returnează adresa locală la care este legat socketul. Dacă nu se specifică nici o adresă locală în constructor, atunci se va returna <code>0.0.0.0</code> .
<code>int getLocalPort()</code>	Returnează numărul portului la care este legat socketul sau -1 dacă acesta nu este legat.

9.3.2. Clasa `Socket`

Un obiect `Socket` corespunde unei conexiuni de rețea TCP. Utilizând această clasă, un client poate stabili un canal de comunicație bazat pe fluxuri cu o mașină de la distanță.

Structura generală a unui client care stabilește o conexiune este foarte simplă. Acesta creează un obiect `Socket` care deschide o conexiune către un server și apoi, prin intermediul aceluiași obiect, se realizează comunicarea cu serverul. Programul client trebuie să cunoască adresa serverului, precum și portul acestuia. Dacă la adresa și portul indicat nu există nici un server activ, atunci stabilirea conexiunii va eșua și va fi aruncată o excepție `IOException`.

Clasa `Socket` este utilizată și de server pentru comunicarea cu clienții. Pentru fiecare client va exista un obiect de acest tip (obținut în urma apelului metodei `accept()` din clasa `ServerSocket`), iar trimiterea, respectiv recepționarea mesajelor pentru server sunt identice cu cele pentru client.

Crearea unui obiect `Socket` va realiza conexiunea cu serverul. Prezintă cei mai semnificativi constructori:

- `Socket()`
creează un obiect `Socket` neconectat.
- `Socket(String gazda, int port)` throws `UnknownHostException`, `IOException`
Creează un obiect `Socket` conectat la portul indicat al adresei gazdă specificată. gazda poate conține numele serverului sau adresa IP. Portul trebuie să fie cuprins între valorile 1 și 65535.
- `Socket(InetAddress adresa, int port)` throws `IOException`
Față de constructorul precedent, se specifică adresa IP a serverului prin intermediul unui obiect de tip `InetAddress`.
- `Socket(String gazda, int port, InetAddress adresaLocală, int portLocal)` throws `IOException`
Creează un obiect `Socket` legat la adresa locală (`adresaLocală`), portul local (`portLocal`) și conectat la portul specificat (`port`) al adresei gazdă specificate (`gazda`). Dacă `adresaLocală` este `null`, atunci se utilizează adresa locală implicită. Dacă `portLocal` este 0, atunci se utilizează un port local arbitrar neutilizat.
- `Socket(InetAddress adresa, int port, InetAddress adresaLocală, int portLocal)` throws `IOException`
Similar cu precedentul constructor, cu deosebirea că primul argument este un obiect din clasa `InetAddress`.

Metodele clasei `Socket` permit identificarea gazdei de la distanță, a portului local și a celui de la distanță, precum și extragerea de fluxuri în vederea comunicării bidirecționale.

Pentru comunicarea într-o conexiune TCP, trebuie mai întâi stabilirea conexiunii (crearea unui obiect `Socket`), apoi apelarea metodelor `getInputStream()` și `getOutputStream()` pentru obținerea fluxurilor de intrare și respectiv ieșire, cu ajutorul cărora se realizează comunicarea la distanță. Așadar, atât clientul, cât și serverul vor avea câte un obiect `InputStream` și `OutputStream` în scopul comunicării.

În tabelul de mai jos sunt date câteva metode uzuale ale clasei `Socket`:

Prototipul metodei	Descrierea metodei
InputStream getInputStream() throws IOException	Returnează un flux de intrare care permite citirea datelor din socket (trimise de celălalt partener).
OutputStream getOutputStream() throws IOException	Returnează un flux de ieșire care permite scrierea datelor în socket (către celălalt partener).
void close() throws IOException	Închide socketul curent, închizând conexiunea și eliberează orice resursă rețea și sistem utilizate.
InetAddress getInetAddress()	Returnează adresa IP a mașinii de la distanță.
int getPort()	Returnează numărul portului gazdei de la distanță la care este conectat socketul curent.
InetAddress getLocalAddress()	Adresa mașinii locale de unde este conectat socketul curent.
int getLocalPort()	Returnează portul local de la care este conectat socketul curent.

9.3.3. O aplicație simplă client/server orientată conexiune

În această secțiune, vom prezenta un exemplu simplu de aplicație client/server orientată conexiune. Aplicația client citește două numere reale de la tastatură, după care le trimite serverului. Acesta calculează maximum dintre cele două numere primite și returnează maximum.

Clasa MaxServer conține definițiile a trei metode statice:

- public static void trimiteDateCatreClient(DataOutputStream out, String sir) throws IOException

Această metodă trimite șirul sir către fluxul de ieșire out utilizând metoda writeUTF().

- public static String primeșteDateDeLaClient(DataInputStream in) throws IOException

Această metodă citește, utilizând metoda readUTF(), un String trimis prin fluxul de intrare in, pe care apoi îl returnează ca rezultat.

- public static void main(String [] args)

Se încearcă crearea unui obiect ServerSocket, numit server, care să asculte la portul 2003. În caz de eșec, se aruncă o excepție IOException. În vederea creării socketului s, se apelează metoda accept() pentru obiectul server, după care se creează cele două fluxuri in și out, asociate socketului s. Apoi, se așteaptă de la client cele două valori double, se calculează maximum acestora, după care se trimite către client. În caz de eșec, se aruncă o excepție IOException.

Fișierul MaxServer.java are următorul conținut:

```
import java.net.*;
import java.io.*;
```

```
/** clasa principala */
public class MaxServer {
    /** metoda pentru trimiterea datelor catre client */
    public static void trimiteDateCatreClient(DataOutputStream
out, String sir) throws IOException {
        out.writeUTF(sir); // trimite catre client sirul
        out.flush(); // goleste fluxul
        // afisarea unui mesaj protected ecran
        System.out.println("Am trimis catre client: " + sir);
    }

    /** metoda pentru receptionarea datelor de la client */
    public static String primeșteDateDeLaClient(DataInputStream
in) throws IOException {
        String sir = in.readUTF(); // obtine raspunsul de la client
        // afisarea unui mesaj la consola
        System.out.println("Am primit de la client: " + sir);
        return sir;
    }

    /** metoda main() */
    public static void main(String [] args) {
        /** declararea variabilelor */
        DataInputStream in = null;
        DataOutputStream out = null;
        Socket s = null;
        ServerSocket server = null;
        try {
            /** crearea socketului pentru server la portul 2003*/
            server = new ServerSocket(2003);
            /** afisarea unui mesaj la consola */
            System.out.println("Așteptam un client...");
            /** acceptarea unei conexiuni cu un client */
            s = server.accept();
            System.out.println(
                "S-a stabilit conexiunea cu clientul.");
            /** obtinerea unui flux de intrare convenabil */
            in = new DataInputStream
                (new BufferedInputStream (s.getInputStream()));
            /** obtinerea unui flux de iesire convenabil */
            out = new DataOutputStream
                (new BufferedOutputStream (s.getOutputStream()));
        } catch (IOException e) {
            /** tratarea exceptiei */

```



```

        System.out.println("Eroare la conectare: " + e);
    }
    /** declararea altor variabile auxiliare */
    String sirNumere = "";
    double nr1 = 0.0, nr2 = 0.0;
    double max = 0.0;
    try {
        /** primirea unui numar */
        sirNumere = primeșteDateDeLaClient(in);
        Double tmp = Double.valueOf(sirNumere);
        nr1 = tmp.doubleValue();
        /** primirea celuiilalt numar */
        sirNumere = primeșteDateDeLaClient(in);
        tmp = Double.valueOf(sirNumere);
        nr2 = tmp.doubleValue();
        /** calcularea maximului */
        max = (nr1 < nr2)? nr2 : nr1;
        /** trimiterea rezultatului */
        trimiteDateCatreClient(out, Double.toString(max));
    } catch (IOException e) {
        /** tratarea cazului de exceptie */
        System.out.println(
            "Eroare la trimitere/primire date: " + e);
    }
} // sfarsitul definitiei metodei main()
} // sfarsitul definitiei clasei MaxServer

```

În mod asemănător, clasa MaxClient conține definițiile a trei metode statice:

- public static void trimiteDateCatreServer(DataOutputStream out, String sir) throws IOException

Această metodă trimite șirul sir către fluxul de ieșire out utilizând metoda writeUTF().

- public static String primeșteDateDeLaServer(DataInputStream in) throws IOException

Această metodă citește, utilizând metoda readUTF(), un String trimis prin fluxul de intrare in, pe care apoi îl returnează ca rezultat.

- public static void main(String [] args)

Se încearcă crearea unui obiect de tip Socket, numit s, asociat mașinii locale (de adresă 127.0.0.1) la portul 2003. În caz de eșec, se aruncă o excepție IOException. Apoi se creează cele două fluxuri in și out, asociate socketului s. Apoi, se așteaptă de la tastatură introducerea a două valori double care se trimit către server apelând metoda trimiteDateCatreServer(). În variabila rezultat se va primi rezultatul returnat de apelul metodei primeșteDateDeLaServer() care va fi afișat la consolă. În caz de eșec, se aruncă o excepție IOException.

Conținutul fișierului MaxClient.java este următorul:

```

import java.net.*;
import java.io.*;

public class MaxClient {
    /** metoda de trimitere a datelor catre server */
    public static void trimiteDateCatreServer(DataOutputStream
        out, String sir) throws IOException {
        out.writeUTF(sir); // trimite catre server sirul
        out.flush(); // golește sirul din flux
        // afisarea unui mesaj la consola
        System.out.println("Am trimis catre server: " + sir);
    }

    /** metoda de primire a datelor de la server */
    public static String primeșteDateDeLaServer(DataInputStream in)
        throws IOException {
        String sir = in.readUTF(); // obtine raspunsul de la server
        System.out.println("Am primit de la server: " + sir);
        return sir;
    }

    /** metoda main() */
    public static void main(String [] args) {
        /** declararea datelor locale */
        DataInputStream in = null;
        DataOutputStream out = null;
        Socket s = null;
        try {
            /** stabilirea unei conexiuni cu
            serverul local de la portul 2003 */
            s = new Socket("127.0.0.1", 2003);
            /** afisarea unui mesaj de succes */
            System.out.println("Ne-am conectat la server.");
            /** obtinerea unui flux de intrare convenabil */
            in = new DataInputStream
                (new BufferedInputStream(s.getInputStream()));
            /** obtinerea unui flux de iesire convenabil */
            out = new DataOutputStream
                (new BufferedOutputStream(s.getOutputStream()));
        } catch (IOException e) {
            /** tratarea exceptiei */
            System.out.println("Eroare la conectare: " + e);
        }
    }
}

```

```

        System.exit(1);
    }
    /** declararea altor date auxiliare */
    double nr1 = 0.0, nr2 = 0.0;
    BufferedReader tastatura;
    String linie;
    try {
        /** obtinerea unui flux de la intrarea standard
            (de la tastatura) */
        tastatura = new BufferedReader(new InputStreamReader(
            System.in));
        System.out.flush();
        /** citirea primului numar de la tastatura */
        System.out.print("Dati primul numar: ");
        linie = tastatura.readLine();
        Double tmp = Double.valueOf(linie);
        nr1 = tmp.doubleValue();
        System.out.flush();
        /** citirea celui de-al doilea numar */
        System.out.print("Dati al doilea numar: ");
        linie = tastatura.readLine();
        tmp = Double.valueOf(linie);
        nr2 = tmp.doubleValue();
        tastatura.close(); // inchiderea intrarii standard
    } catch (IOException e) {
        /** tratarea cazului de exceptie */
        System.out.println("Citire gresita de la tastatura: " + e);
    }
    /** Comunicarea cu serverul */
    String rezultat = "";
    try {
        /** trimiterea datelor */
        trimiteDateCatreServer(out, Double.toString(nr1));
        trimiteDateCatreServer(out, Double.toString(nr2));
        /** primirea rezultatului */
        rezultat = primeșteDateDeLaServer(in);
    } catch (IOException e) {
        System.out.println(
            "Eroare la trimitere/primire date: " + e);
    }
    /** Afisarea rezultatului */
    System.out.println("Rezultat: " + rezultat);
} // sfarsitul definitiei metodei main()
} // sfarsitul definitiei clasei MaxClient

```

Cele două programe Java sunt compilate și executate cu ajutorul comenzilor `javac`, respectiv `java`. Mai întâi se pornește în execuție serverul, după care clientul. Pentru client va trebui să introducem de la tastatură două numere.

O posibilă execuție a clientului poate fi:

```

Ne-am conectat la server.
Dati primul numar: 123.230
Dati al doilea numar: -0.07
Am trimis catre server: 123.23
Am trimis catre server: -0.07
Am primit de la server: 123.23
Rezultat: 123.23

```

iar a serverului:

```

Asteptam un client...
S-a stabilit conexiunea cu clientul.
Am primit de la client: 123.23
Am primit de la client: -0.07
Am trimis catre client: 123.23

```

În acest exemplu, serverul acceptă o conexiune, deservește cererea, după care totul se termină. În practică, la apariția unui client se creează un nou fir de execuție care deservește cererea, iar firul principal se ocupă doar de acceptarea conexiunilor. De regulă, un server se execută la infinit, eventual acesta poate primi anumite comenzi de la tastatură pentru oprire, reinițializare sau repornire.

9.4. Programarea rețelor prin intermediul datagramelor

În Java, socketurile pentru protocoale fără conexiune (în speță, UDP) sunt implementate de clasele `DatagramPacket` și `DatagramSocket`.

9.4.1. Clasa `DatagramPacket`

Clasa `DatagramPacket` corespunde unui pachet de date numit și *datagramă*. Aceasta permite specificarea adresei destinatarului și a informației transportate. Este asigurată primirea mesajului de către destinatar. O aplicație care recepționează un astfel de mesaj poate extrage din respectivul pachet atât informația trimisă, cât și adresa expeditorului (adresa serverului și numărul portului).

Există două tipuri de constructori pentru clasa `DatagramPacket` după scopul de întrebuințare a pachetelor: pentru transmitere sau pentru recepționare. Constructorii pentru recepționarea datagramelor sunt:

- `DatagramPacket(byte[] buf, int lungime)`

Parametrul `buf` desemnează zona de memorie în care se va copia conținutul mesajului (aceasta trebuie să fie alocată), iar `lungime`, numărul maxim pe care îl poate avea respectivul mesaj. Trebuie să avem grijă ca valoarea dată de `lungime` să nu depășească capacitatea lui `buf` (`buf.length`).

- `DatagramPacket(byte[] buf, int deplasament, int lungime)`

Ca și mai sus, parametrul `buf` este utilizat pentru a indica zona de memorie în care se va salva mesajul primit. Acesta va fi stocat în respectivul buffer începând cu poziția `deplasament` și va conține cel mult `lungime` octeți. De asemenea, trebuie să avem grijă să nu se depășească capacitatea bufferului.

În plus, pentru transmiterea datagramelor trebuie să mai indicăm adresa destinatarului. Prin urmare, constructorii pentru trimiterea datagramelor vor avea în plus față de cei pentru recepționare doi parametri: unul pentru adresa mașinii și unul pentru indicarea portului serverului. Cei mai importanți constructori ai clasei `DatagramPacket` utilizați pentru transmiterea datagramelor sunt:

- `DatagramPacket(byte[] buf, int lungime, InetAddress adresa, int port)`
- `DatagramPacket(byte[] buf, int deplasament, int lungime, InetAddress adresa, int port)`

Parametrii `buf`, `lungime` și `deplasament` au aceeași semnificație ca la constructorii pentru primirea datagramelor, cu deosebirea că se referă la transmiterea mesajului (`lungime` va indica numărul de caractere pe care îl va avea mesajul). În plus, parametrul `adresa` indică adresa mașinii serverului, iar `port` specifică portul la care așteaptă datagrama respectivul server.

Pachetul va fi trimis către portul specificat și gazda specificată. Trebuie să existe un server UDP care ascultă la portul specificat pentru trimiterea pachetelor. **Atenție!** Numerele de port UDP și TCP sunt complet independente, deci putem avea și un server UDP, și un server TCP care ascultă la același port.

În funcție de tipul constructorului apelat se va trimite, respectiv recepționa o datagramă. Dacă pachetul este primit, atunci adresa corespunde gazdei sursă, iar dacă pachetul a fost creat pentru transmisie, atunci adresa corespunde mașinii destinație. Conținutul și adresa unui obiect `DatagramPacket` pot fi interogate și modificate cu metodele:

Prototipul metodei	Descrierea metodei
<code>InetAddress getAddress()</code>	Returnează adresa IP a datagramei.
<code>int getPort()</code>	Returnează portul conținut de datagramă.
<code>byte[] getData()</code>	Returnează conținutul datagramei.
<code>int getLength()</code>	Întoarce dimensiunea mesajului datagramei (numărul de octeți).
<code>void setAddress(InetAddress adresa)</code>	Stabilește adresa pentru datagrama curentă.
<code>void setData(byte[] buf)</code>	Modifică conținutul mesajului.
<code>void setLength(int lungime)</code>	Modifică dimensiunea mesajului.

9.4.2. Clasa `DatagramSocket`

Clasa `DatagramSocket` corespunde unui socket pentru datagrama, prin intermediul căruia se vor putea primi, respectiv recepționa astfel de pachete. Această clasă se poate utiliza atât pentru trimiteri, cât și pentru primiri de datagrama (obiecte `DatagramPacket`). Ca și în cazul unui server TCP, unul UDP trebuie să asculte la un număr de port particular între 1 și 65535 (porturile cuprinse între 1 și 1024 sunt rezervate, de obicei, pentru aplicații sistem). Deoarece UDP nu este orientat conexiune, se va crea un singur socket (obiect `DatagramSocket`) pentru trimiterea pachetelor către diferite destinații și primirea pachetelor de la diferite surse.

Un obiect `DatagramSocket` va recepționa la un anumit port UDP de pe mașina locală pachetele care sosesc. Numărul portului se poate specifica sau eventual lăsa sistemul de operare să-l asigneze. De obicei, serverul va alege un port particular cu care să opereze, iar clienții vor permite stabilirea unui port aleator. Numărul de port și adresa expeditorului vor fi automat inserate în fiecare pachet trimis. Există și posibilitatea legării unui obiect `DatagramSocket` la o adresă locală particulară pe o mașină a căreia îi sunt asignate mai multe adrese IP.

Cei mai utilizați constructori ai clasei `DatagramSocket` sunt:

- `DatagramSocket()` throws `SocketException`

Se creează un obiect `DatagramSocket` cu un număr de port ales aleator.

- `DatagramSocket(int port)` throws `SocketException`

Acest constructor creează un socket UDP care ascultă la numărul de port specificat.

- `DatagramSocket(int port, InetAddress locala)`
throws `SocketException`

Se creează un obiect `DatagramSocket` care ascultă la numărul de port specificat și este legat la adresa locală indicată. Pentru mașinile cărora le sunt asignate mai multe adrese IP, pachetele sunt trimise utilizând adresa IP specificată.

Clasa `DatagramSocket` furnizează metode pentru trimiterea și primirea de obiecte `DatagramPacket`, închiderea socketului, obținerea adresei locale, setarea timpului de așteptare pentru primire etc.:

Prototipul metodei	Descrierea metodei
<code>void send(DatagramPacket packet)</code> throws <code>IOException</code>	Trimite datagrama pachet la destinatar. În cazul în care nu se cunoaște destinatarul, va apărea o excepție <code>IOException</code> .
<code>void receive(DatagramPacket packet)</code> throws <code>IOException</code>	Recepționează o singură datagramă în obiectul pachet. Ulterior se poate cerceta cine este expeditorul, care este lungimea mesajului etc. Execuția firului de execuție este blocată până când se recepționează un pachet sau se scurge timpul de așteptare.
<code>InetAddress getLocalAddress()</code>	Returnează adresa locală la care este legat socketul.

Prototipul metodei	Descrierea metodei
<code>int getLocalPort ()</code>	Întoarce portul la care așteaptă socketul curent.
<code>void close ()</code>	Închide socketul eliberând astfel resursele ocupate.
<code>void setSoTimeout (int timpDeAșteptare) throws SocketException</code>	Setează timpul de așteptare în milisecunde a unui socket la valoarea specificată. Metoda <code>receive ()</code> se va bloca doar pentru timpul de așteptare specificat pentru primirea unui pachet UDP, după care va arunca o excepție <code>InterruptedException</code> . Setarea timpului la 0 va dezactiva această facilitare.
<code>int getSoTimeout () throws SocketException</code>	Returnează timpul de așteptare a socketului curent sau 0 dacă acesta nu este setat.
<code>void setSendBufferSize (int lungime) throws SocketException</code>	Setează lungimea bufferului socketului pentru trimitere la valoarea specificată. Pachetele mai mari de această valoare nu vor putea fi trimise.
<code>int getSendBufferSize () throws SocketException</code>	Returnează lungimea curentă a bufferului socketului pentru trimitere.
<code>void setReceiveBufferSize (int lungime) throws SocketException</code>	Setează lungimea bufferului socketului pentru recepționare la valoarea specificată. Pachetele mai mari de această valoare nu vor putea fi primite.
<code>int getReceiveBufferSize () throws SocketException</code>	Returnează lungimea curentă a bufferului socketului pentru recepționare.
<code>void connect (InetAddress adresa, int port) throws SocketException</code>	Conectează acest socket la adresa și portul specificat la distanță. Schimbul de pachete se va realiza doar cu socketul de la adresa și portul respectiv. În general, această metodă se utilizează din motive de performanță și nu este cerută pentru operațiunile uzuale.
<code>void disconnect ()</code>	Deconectează socketul conectat.
<code>InetAddress getInetAddress ()</code>	Returnează obiectul <code>InetAddress</code> la care este conectat socketul sau <code>null</code> dacă acesta nu este conectat.
<code>int getPort ()</code>	Întoarce numărul portului la care este conectat socketul sau -1 dacă acesta nu este conectat.

La crearea unui obiect `DatagramSocket` sau la setarea acestuia se poate arunca o excepție `SocketException`, iar dacă apar probleme la trimiterea sau primirea pachetelor, se va arunca excepția `IOException`.

Managerul de securitate (*SecurityManager*) poate restricționa accesul transmisiunilor și primirilor datagramelor. De exemplu, appleturile nesigure nu pot trimite și nici primi date de la alt server decât serverul inițial. Restricțiunile de primire sunt rezolvate prin verificarea adresei sursă a fiecărui pachet primit. Dacă un pachet vine dintr-o sursă nevalidă, atunci acesta este eliminat și metoda `receive ()` continuă să aștepte pachete valide.

9.4.3. O aplicație simplă client/server neorientată conexiune

Comunicarea cu un protocol fără conexiune este mai simplă decât cu unul orientat conexiune, deoarece atât clientul, cât și serverul recurg la obiecte de tip `DatagramSocket`. De regulă, un server care lucrează cu datagrame trebuie să parcurgă următoarele etape:

- crearea unui obiect de tip `DatagramSocket` asociat cu un număr de port specificat;
- crearea unui obiect `DatagramPacket`;
- cu ajutorul obiectului de tip `DatagramSocket` se stochează o datagramă în obiectul de tip `DatagramPacket`;
- se procesează cererea;
- se completează o nouă datagramă;
- se trimite datagrama completată anterior.

La partea client, ordinea este oarecum inversată: mai întâi se trimite o datagramă, după care se așteaptă recepționarea răspunsului.

Vom aplica șablonul de mai sus în vederea scrierii unei aplicații client/server care utilizează datagramele. Clientul va cere serverului timpul curent și serverul îl va furniza.

Pentru server am creat clasa `TimpServer`. Metoda `main ()` începe prin crearea unui obiect `DatagramSocket` care utilizează portul 2003. Apoi creăm un obiect `DatagramPacket` care va conține datele primite prin intermediul obiectului de tip `DatagramSocket`. Serverul posedă o buclă infinită în care primește cereri de la clienți utilizând metoda `receive ()` pentru obiectul `DatagramSocket` și apoi trimite răspunsuri.

Codul sursă al clasei `ServerTimp` este:

```
import java.net.*;
import java.io.*;

public class TimpServer {
    /** declararea unei date membre */
    static DatagramSocket socket;

    public static void main(String[] argv) {
        try {
            /** crearea unui socket UDP la portul 2003*/
            socket = new DatagramSocket(2003);
        } catch (SocketException e) {
            System.err.println("Nu putem crea socketul" +
                               e.getMessage());
            System.exit(1);
        }
    }
}
```

```

/** pregatirea pentru receptionarea datagramelor */
DatagramPacket datagrama;
datagrama = new DatagramPacket(new byte[1], 1);
System.out.println("Asteptam clienti...");
/** intrarea in bucla infinita */
while (true) {
    try {
        /** receptionarea unei datagrame */
        socket.receive(datagrama);
        /** raspunde la cererea primita */
        raspunde(datagrama);
    } catch (IOException e) {
        System.err.println(
            "Nu putem primi sau trimite datagrama"
            + e.getMessage());
    }
} // sfarsitul definitiei metodei main()

/** metoda pentru raspunderea la cereri */
static void raspunde(DatagramPacket cerere) {
    /** obtinerea unui flux de iesire convenabil */
    ByteArrayOutputStream baos;
    baos = new ByteArrayOutputStream();
    DataOutputStream out = new DataOutputStream(baos);
    /* scrierea in flux a timpul curent in milisecunde */
    try {
        out.writeLong(System.currentTimeMillis());
    } catch (IOException e) {
        System.err.println("Nu putem trimite timpul curent"
            + e.getMessage());
    }
    /** construirea raspunsului */
    DatagramPacket raspuns;
    /** stabilirea mesajului */
    byte[] data = baos.toByteArray();
    /** crearea datagramei */
    raspuns = new DatagramPacket(data, data.length,
        cerere.getAddress(), cerere.getPort());
    /** trimiterea efectiva a pachetului */
    try {
        socket.send(raspuns);
    }
    catch (IOException e) {
        System.err.println("Nu putem trimite datagrama"

```

```

        + e.getMessage());
    }
    // sfarsitul definitiei metodei raspunde()
} // sfarsitul definitiei clasei TimpServer

```

Metoda raspunde() se ocupă de trimiterea răspunsurilor. Vom scrie timpul curent ca o valoare long într-un șir de octeți. Apoi, pregătim trimiterea șirului de octeți prin crearea unui obiect DatagramPacket care încapsulează șirul, adresa și numărul de port al clientului care solicită timpul curent. În final, se trimit datele apelând metoda send() a obiectului DatagramSocket.

Codul sursă corespunzător aplicației client este:

```

import java.net.*;
import java.io.*;
import java.util.*;

public class TimpClient {
    public static void main(String argv[]) {
        if (argv.length != 1) {
            System.out.println(
                "Dati comanda: ClientTimp numeServer");
            System.exit(1);
        }
        DatagramSocket socket = null;
        try {
            socket = new DatagramSocket();
        } catch (SocketException e) {
            System.err.println("Nu putem crea socketul" +
                e.getMessage());
            System.exit(1);
        }
        long timp = 0;
        try {
            byte[] buffer = new byte[1];
            socket.send(new DatagramPacket(buffer, 1,
                InetAddress.getByName(argv[0]), 2003));
            DatagramPacket raspuns = new DatagramPacket(new byte[8], 8);
            socket.receive(raspuns);
            ByteArrayInputStream baos;
            baos = new ByteArrayInputStream(raspuns.getData());
            DataInputStream dis = new DataInputStream(baos);
            timp = dis.readLong();
        } catch (IOException e) {
            System.err.println("Nu putem trimite sau primi datagrama"
                + e.getMessage());
        }
    }
}

```

```

    }
    System.out.println("Timpul primit de la server este "
        + new Date(timp));
    socket.close();
} // sfarsitul definitiei metodei main()
} // sfarsitul definitiei clasei ClientTimp

```

Pentru client am implementat clasa `ClientTimp`. Acesta verifică mai întâi numărul corect de argumente (trebuie să fie un argument cu adresa IP). Apoi se creează un obiect `DatagramSocket` pentru comunicarea cu serverul. Urmează crearea unui obiect `DatagramPacket` care va conține cererea ce trebuie trimisă către server utilizând metoda `send()`. Apoi, creăm încă un obiect `DatagramPacket` pentru primirea răspunsului de la server cu ajutorul apelului `receive()`. În final, clientul afișează timpul curent și închide socketul (indiferent dacă se aruncă o excepție sau nu).

9.4.4. Clasa `InetAddress`

Datele pot fi trimise prin rețea, utilizând adresa IP specifică mașinii de destinație. Clasa `InetAddress` furnizează acces abstract la adresele IP. Avantajul utilizării acestei clase pentru reprezentarea unei adrese IP într-un întreg pe 32 biți este că aplicațiile vor fi transparente portabile la IPv6 (IP versiunea 6), care furnizează adrese pe 128 de biți.

Nu există constructori pentru această clasă. Instanțele trebuie realizate prin intermediul metodelor statice `getByName()`, `getLocalHost()` și `getAllByName()`. Acestea au sintaxa:

- `InetAddress getLocalHost()` throws `UnknownHostException`
Aceasta metodă returnează un obiect de tip `InetAddress` corespunzător mașinii locale.
- `InetAddress getByAddress(byte[] addr)`
Întoarce un obiect de tip `InetAddress` corespunzător adresei de IP date în tabloul specificat. Pentru adresele IPv4 se va transmite un tablou de 4 elemente (octeți), iar pentru IPv6, de 16 elemente.
- `InetAddress getByName(String host)` throws `UnknownHostException`

Pentru mașina gazdă specificată se construiește un obiect de tip `InetAddress`. Host-ul poate fi specificat prin nume (de exemplu, `dil.info.uaic.ro`) sau prin adresa IP (de exemplu, `193.231.30.182`).

- `InetAddress [] getAllByName (String host)` throws `UnknownHostException`
Returnează un vector de obiecte `InetAddress` corespunzător fiecărei adrese IP cunoscute pentru host-ul specificat. De obicei, siturile Web de trafic înalt au mai multe adrese IP pentru un singur nume de host.

Există două excepții care pot fi aruncate de metodele statice de mai sus:

- `UnknownHostException`, care este subclasă a clasei `IOException` și indică faptul că host-ul căutat nu a fost identificat cu succes;

- `SecurityException` - poate fi aruncată dacă managerul de securitate nu permite o operație specifică. Un applet (de neîncredere) poate doar să construiască un obiect `InetAddress` pentru numele gazdei corespunzătoare serverului Web de unde provine codul său.

Clasa `InetAddress` are următoarele metode instanță (adică asociate unui obiect al clasei, și nu clasei):

Prototipul metodei	Descrierea metodei
<code>byte [] getAddress()</code>	Întoarce un vector de octeți corespunzător adresei IP. Vectorul este în ordinea octeților adresei rețelei, adică octetul semnificativ primul. Pentru IPv4, acest vector are doar 4 octeți, iar pentru IPv6 are 16 octeți.
<code>String getHostName()</code>	Returnează numele mașinii gazde. În cazul în care nu se poate afla numele, se întoarce adresa de IP.
<code>String getAddress()</code>	Se întoarce adresa IP.
<code>String toString()</code>	Convertește adresa IP într-un șir de caractere.

Exemplul 9.4.1. Următorul program Java afișează adresa IP locală.

```

import java.net.*;

public class adresaIPLocala {
    public static void main(String args[]) {
        InetAddress adresaIPLocala = null;
        try {
            adresaIPLocala = InetAddress.getLocalHost();
            System.out.println("adresaIPLocala = " + adresaIPLocala);
        } catch (UnknownHostException e) {
            System.err.println("Nu putem gasi gazda: " + e.toString());
        }
    }
}

```

O posibilă execuție a acestui program poate fi:

```
adresaIPLocala = felix/127.0.0.1
```

unde `felix` este numele calculatorului, iar `127.0.0.1` indică faptul că este vorba de mașina locală.

Exemplul 9.4.2. Programul Java afișează adresa IP a unei mașini la distanță, plecând de la numele de domeniu.

```

import java.net.*;
import java.io.*;

```

```

public class adresaIPLaDistanta {
    public static void main(String args[]) {
        BufferedReader tastatura;
        tastatura = new BufferedReader(new InputStreamReader(
            System.in),1);
        System.out.print("Dati un nume de domeniu: ");
        String linie = "";
        // citim numele de domeniu reprezentat ca un String
        try {
            System.out.flush();
            linie = tastatura.readLine();
            tastatura.close();
        } catch (IOException e) {
            System.out.println("Exceptie la citirea de la tastatura"
                + e);
            System.exit(2);
        }
        InetAddress adresaIP = null;
        try {
            adresaIP = InetAddress.getByName(linie);
            System.out.println(adresaIP);
        } catch (UnknownHostException e) {
            System.out.println("Nu putem gasi gazda: " + e);
        }
    }
}

```

O posibilă execuție a acestui program este:

```

Dati un nume de domeniu: webgroup.info.uaic.ro
webgroup.193.231.30.227

```

9.4.5. Clasa URL

Clasa URL furnizează acces de nivel înalt la datele de la distanță. Un obiect de tip URL încapsulează o specificare a unui locator uniform de resurse (URL). O dată creat un astfel de obiect, acesta poate fi utilizat pentru accesarea datelor locației indicate. Un URL permite accesul la date fără a fi nevoie de detaliile protocolului folosit, cum ar fi HTTP sau FTP. Pentru anumite tipuri de date, un obiect de tip URL furnizează un mod de a obține date deja încapsulate în alte obiecte asemănătoare. De exemplu, un URL poate furniza date JPEG încapsulate într-un obiect ImageProducer sau date text încapsulate într-un obiect String.

Cel mai utilizat constructor pentru clasa URL este cel cu un parametru de tip String care conține adresa URL corespunzătoare. În cazul în care parametrul nu conține un URL valid, se va arunca excepția MalformedURLException.

De exemplu, instanțierea clasei URL se poate realiza astfel:

```

try {
    URL sitFacultate = new URL(
        "http://www.infoiasi.ro/fcs/index.html");
} catch (MalformedURLException e) {
    System.err.println("URL-gresit: " + e.getMessage());
}

```

O dată creat un obiect de tip URL, se pot utiliza metodele sale pentru obținerea informațiilor încapsulate:

Prototipul metodei	Descrierea metodei
String getFile()	Returnează numele fișierului.
String getHost()	Extrage numele mașinii gazdă.
String getPort()	Întoarce portul utilizat.
String getProtocol()	Returnează numele protocolului.
String getRef()	Întoarce referința (sau ancora).
String getQuery()	Extrage șirul de interogare.
boolean sameFile (URL altul)	Testează dacă obiectul curent și cel specificat indică aceeași resursă (fișier).
URLConnection openConnection() throws IOException	Realizează o conexiune la resursa indicată.
final InputStream openStream() throws IOException	Deschide o conexiune la adresa referită și întoarce obiectul InputStream corespunzător.
final Object getContent()	Returnează conținutul resursei referite sub forma unui obiect al cărui tip este convenabil ales în funcție de tipul fișierului.

Exemplul 9.4.3. Appletul de mai jos citește o adresă URL și întoarce numele protocolului, numele mașinii gazdă, portul, numele fișierului și referința (ancora).

```

import java.applet.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class InfoURL extends Applet implements ActionListener {
    /** datele membre */
    URL url;
    String sirURL;
    Button afiseaza;
}

```

```

TextField numeCamp;

/** metoda de initializare */
public void init() {
    /** crearea unui buton */
    afiseaza = new Button("Dati adresa URL");
    add(afiseaza);
    afiseaza.addActionListener(this);
    /** crearea unui camp de text */
    numeCamp = new TextField(60);
    add(numeCamp);
}

/** implementarea metodei din interfata ActionListener */
public void actionPerformed(ActionEvent event) {
    /** preluam URL introdus in campul text */
    sirURL = numeCamp.getText();
    try {
        /** instantiem un obiect de tip URL */
        url = new URL(sirURL);
        /** obtinem numele fisierului */
        String numeFisier = url.getFile();
        System.out.println("Numele fisierului este " + numeFisier);
        /** obtinem numele masinii gazda */
        String numeHost = url.getHost();
        System.out.println("Numele host-ului este " + numeHost);
        /** obtinem numele portului */
        int numarPort = url.getPort();
        System.out.println("Numarul portului este " + numarPort);
        /** obtinem numele protocolului */
        String numeProtocol = url.getProtocol();
        System.out.println("Numele protocolului este " +
            numeProtocol);
        /** extragem referinta (ancora) */
        String numeReferinta = url.getRef();
        System.out.println("Numele referintei este " +
            numeReferinta);
    } catch (MalformedURLException e) {
        /** tratarea exceptiei */
        System.err.println("Eroare: " + e.getMessage());
    }
}

```

Dacă se va introduce în câmpul text al appletului adresa: <http://www.infoiasi.ro:8080/fcs/index.html>, atunci se va afișa la consolă:

```

Numele fisierului este /fcs/index.html
Numele host-ului este www.infoiasi.ro
Numarul portului este 8080
Numele protocolului este http
Numele referintei este null

```

Continuăm cu citirea unui fișier de pe un server aflat la distanță care se realizează în mod similar cu citirea unui fișier local (aceasta datorită puterii bibliotecilor Java). În loc de numele fișierului vom specifica adresa URL a acestuia.

Pentru a accesa conținutul unui fișier linie cu linie în loc să-l obținem încapsulat într-un obiect, putem utiliza metoda `openStream()` a clasei `URL` sau metoda `getInputStream()` din clasa `URLConnection` care întoarce o referință la un obiect din clasa `InputStream`.

Pentru crearea unui obiect din clasa `URLConnection` se utilizează metoda `openConnection()` din clasa `URL`. Dacă nu este deja deschisă o conexiune, metoda `openConnection()` deschide o conexiune prin apelarea metodei `openConnection()` din clasa `URLStreamHandler` pentru acest URL. Un obiect `URLStreamHandler` pentru protocolul URL este creat de constructorul URL. În pachetul `java.net`, subclasele clasei `URLStreamHandler` se ocupă cu diferite protocoale pentru comunicație.

Exemplul 9.4.4. Programul Java de mai jos citește de la tastatură o adresă URL și afișează conținutul fișierului pe ecran. Aplicația permite doar afișarea documentelor de tip text.

```

import java.io.*;
import java.net.*;

public class ContinutTextURL {
    /** declararea datelor membre */
    private BufferedReader tastatura, fluxIntrare;

    /** metoda principala */
    public static void main(String[] args) {
        ContinutTextURL continut = new ContinutTextURL();
        continut.afisare();
    }

    /** metoda pentru afisarea continutului de la adresa data de
        URL */
    private void afisare() {
        /** declararea variabilelor locale */
        String sirURL = "";
        String linie = "";
    }
}

```



```

/** obtinerea unui flux de la tastatura */
tastatura = new BufferedReader(new InputStreamReader
(System.in), 1);
try {
    /** obtinerea adresei URL de la tastatura */
    sirURL = prompt("Dati o adresa URL (ex.: " +
        "http://www.infoiasi.ro/):");
    /** obtinerea unei instante a clasei URL */
    URL adresaURL = new URL(sirURL);
    /** crearea unei conexiuni */
    URLConnection conexiune = adresaURL.openConnection();
    /** obtinerea unui flux pentru citirea continutului
        resursei */
    fluxIntrare = new BufferedReader(new
        InputStreamReader(conexiune.getInputStream()));
    /** cat timp putem citi cate o linie, o afisam la consola */
    while ((linie = fluxIntrare.readLine()) != null) {
        System.out.println(linie);
    }
} /** tratarea exceptiilor */
catch (MalformedURLException e) {
    System.err.println("URL gresit: " + sirURL + "\n" + e);
    System.exit(2);
} catch (IOException e) {
    System.err.println("Eroare la stabilirea conexiunii: " + e);
    System.exit(1);
}
}

/** metoda pentru citirea unei linii de la tastatura */
private String prompt(String mesaj) {
    String raspuns = "";
    try {
        /** afisarea unui mesaj */
        System.out.print(mesaj);
        /** golirea fluxului de iesire */
        System.out.flush();
        /** citirea unei linii de la tastatura */
        raspuns = tastatura.readLine();
    } catch (IOException e) {
        System.err.println("Eroare la citirea de la tastatura:"
            + e);
        System.exit(2);
    }
}
/** intoarcerea rezultatului */

```

```

        return raspuns;
    }
}

```

Dacă indicăm adresa `http://localhost/salut.html`, vom obține conținutul fișierului `salut.html` de pe serverul Web local, care poate avea următorul conținut:

```

<html>
<head>
    <title>Salutari</title>
</head>
<body>
    Salutari de pe serverul Web.
</body>
</html>

```

Cel mai înalt nivel de funcționalitate disponibil de la un obiect de tip URL este furnizat de metoda `getContent()`. Aceasta încearcă să determine tipul de date din fișierul specificat de URL și apoi acesta returnează conținutul încapsulat într-un obiect convenabil pentru acel tip de dată. De exemplu, dacă fișierul conține date în format GIF (cu extensia `.gif`), metoda `getContent()` returnează un obiect din clasa `ImageProducer`. Dacă tipul de date nu este explicit specificat, metoda va încerca să determine tipul după extensia fișierului și posibil, de asemenea, din conținutul acestuia (în general, din primele linii ale sale).

Numele tipurilor de date pe care Java le utilizează sunt conforme cu bine cunoscuta schemă pentru tipuri de date: MIME. Acestea sunt de forma `tip/subtip`.

În tabelul de mai jos se găsesc câteva extensii de fișier recunoscute de Java și tipul MIME al acestora:

Extensie fișier	Tipul MIME
.au	audio/basic
.c	text/plain
.cpp	text/plain
.dvi	application/x-dvi
.exe	application/octet-stream
.gif	image/gif
.html	text/html
.java	text/plain
.txt	text/plain
.zip	application/zip

Exemplul 9.4.5. Vizualizarea textului și a imaginilor din fișiere.

```

import java.applet.*;
import java.net.*;
import java.awt.*;

```

```

import java.awt.image.*;
import java.io.*;

public class VizualizareTextSiImagine extends Applet {
    /** declararea datelor membre */
    Image img;
    String str;

    /** metoda de initializare a appletului */
    public void init() {
        try {
            /** crearea unui URL pentru o imagine */
            URL fisierImagine = new URL(getDocumentBase(), "t1.gif");
            /** obtinerea continutului grafic */
            img = this.createImage((ImageProducer) fisierImagine.
                getContent());
            /** crearea unui URL pentru un fisier text */
            URL fisierText = new URL(getDocumentBase(), "a.txt");
            /** obtinerea continutului */
            str = fisierText.getContent().toString();
        } /** tratarea exceptiilor */
        catch (MalformedURLException e) {
            System.err.println("Eroare:" + e.getMessage());
        } catch (IOException e) {
            System.err.println("Eroare:" + e.getMessage());
        }
        /** actualizarea appletului */
        repaint();
    }

    /** metoda pentru desenarea appletului */
    public void paint(Graphics g) {
        /** desenarea imaginii */
        g.drawImage(img, 0, 0, this);
        /** desenarea continutului text al fisierului */
        g.drawString(str, 10, 200);
    }
}

```

Dacă numele fișierului nu se termină cu o extensie cunoscută, sunt examinați primii octeți. Dacă aceștia „se potrivesc” cu semnătura unui tip cunoscut, atunci se presupune că respectivul document este de acel tip.

Câteva combinații de octeți care sunt recunoscute sunt:

```

GIF8 pentru image/gif
#def pentru image/x-bitmap

```

```

! XPM2 pentru image/x-pixmap
<html> pentru text/html

```

9.4.6. Obținerea unei pagini Web prin intermediul unui socket

Pentru a stabili o conexiune HTTP trebuie să creăm un socket pentru comunicare, să-l conectăm la serverul Web (trebuie să specificăm numele serverului sau adresa de IP și portul acestuia, implicit fiind 80) și să-i trimitem mesaje conforme cu specificațiile protocolului HTTP. Astfel, serverul va putea descifra cererile emise, iar ca rezultat vom obține mesaje HTTP. Acestea sunt alcătuite din antet (eng. *header*) și conținutul mesajului. Aceste două părți sunt despărțite de o linie vidă.

Exemplul 9.4.6. Aplicația va aștepta citirea URL-urilor, iar după ce am specificat unul de la tastatură, va fi afișat, în cazul în care acesta este valid, conținutul paginii Web.

```

import java.net.*;
import java.io.*;

public class ObținePaginaWeb {
    /** declararea datelor membre */
    protected String gazda, fisier;
    protected int port;
    protected Writer iesire;
    protected BufferedReader intrare;

    /** constructorul primește ca argument un String */
    public ObținePaginaWeb(String textURL) throws IOException {
        imparte(textURL);
    }

    /** Metoda imparte() va crea un obiect URL din care putem
        extrage gazda, portul și numele fișierului */
    protected void imparte(String textURL) throws MalformedURLException
        Exception {
        /** obtinerea unui obiect de tip URL */
        URL url = new URL(textURL);
        /** aflarea gazdei */
        gazda = url.getHost();
        /** obtinerea portului */
        port = url.getPort();
        if (port == -1) port = 80;
        /** obtinerea numelui fișierului */
        fisier = url.getFile();
    }
}

```

```

/** metoda pentru conectare */
public void conectare() throws IOException {
    /** crearea unui socket pentru comunicare */
    Socket socket = new Socket(gazda, port);
    /** obtinerea fluxului pentru scrierea in socket */
    OutputStream iesire = socket.getOutputStream();
    /** stabilirea ultimelor caractere din flux */
    this.iesire = new OutputStreamWriter(iesire, "latin1");
    /** obtinerea fluxului pentru citirea din socket */
    InputStream intrare = socket.getInputStream();
    /** stabilirea ultimelor caractere din flux */
    Reader temp = new InputStreamReader(intrare, "latin1");
    this.intrare = new BufferedReader(temp);
}

protected void aduce() throws IOException {
    /** formulam o cerere HTTP 1.0 catre masina de la distanta */
    iesire.write("GET " + fisier + " HTTP/1.0\r\n\n");
    iesire.flush();

    /* afisarea la consola a raspunsului serverului */
    PrintWriter consola = new PrintWriter(System.out);
    String linie;
    while ((linie = intrare.readLine()) != null)
        consola.println(linie);
    /** golirea fluxului aferent consolei */
    consola.flush();
}

/** inchiderea conexiunii cu serverul */
protected void deconectare() throws IOException {
    intrare.close();
}

/** conectarea la server, aducerea paginii Web si deconectarea
    de la server */
public void incarca() throws IOException {
    conectare();
    try {
        aduce();
    }
    finally {
        deconectare();
    }
}

```

```

/** metoda principala */
public static void main(String [] args) throws IOException {
    /** static FileDescriptor.in este un pointer catre fluxul
        de intrare standard. */
    Reader tastatura = new FileReader(FileDescriptor.in);
    BufferedReader buferTastatura = new BufferedReader(tastatura);
    /** citim URL-uri pana apasam combinatia de taste CTRL+C */
    while (true) {
        String textURL;
        System.out.print("Dati un URL: ");
        /** daca nu am apasat CTRL+C iesim din bucla while */
        if ((textURL = buferTastatura.readLine()) == null)
            break;
        try {
            /** crearea unei noi instante */
            ObtinePaginaWeb pag = new ObtinePaginaWeb(textURL);
            /** afisarea continutului paginii */
            pag.incarca();
        } /** tratarea exceptiei */
        catch (IOException ex) {
            ex.printStackTrace();
            /** trecerea la iteratia urmatoare */
            continue;
        }
        /** afisarea unui mesaj */
        System.out.println("- OK -");
    }
    /** afisarea unui mesaj la terminarea aplicatiei */
    System.out.println("Am terminat!");
}

```

O execuție posibilă ar putea fi:

```

Dati un URL: http://localhost/salut.html
HTTP/1.1 200 OK
Date: Sat, 14 Dec 2002 17:11:27 GMT
Server: Apache/1.3.23 (Win32)
Last-Modified: Sat, 14 Dec 2002 15:51:56 GMT
ETag: "0-6a-3dfb539c"
Accept-Ranges: bytes
Content-Length: 106
Connection: close
Content-Type: text/html

```

```

<html>
<head>
  <title>Salutari</title>
</head>
<body>
  Salutari de pe serverul Web.
</body>
</html>
- OK -
Dati un URL: Am terminat!

```

Observăm că este afișat și antetul mesajului HTTP primit. După acesta urmează ca delimitator un rând vid, iar apoi conținutul paginii Web.

9.5. Apelul metodelor la distanță

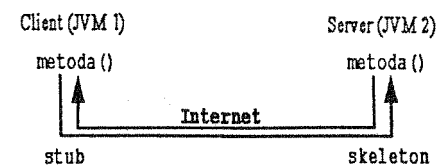
Apelul metodelor la distanță (eng. *Remote Method Invocation* – RMI) furnizează posibilitatea obiectelor Java, care se află pe diferite mașini virtuale, să comunice utilizând apeluri de metode normale. Din moment ce gazdele (calculatoare legate la Internet) pot comunica prin protocolul TCP/IP, înseamnă că aplicațiile de rețea pot fi dezvoltate fără fluxuri și socketuri. Aceasta permite programatorului să evite protocoalele de comunicare complexe dintre aplicații și să adopte, astfel, un stil bazat pe metode de nivel înalt.

Scopul implementării apelului metodelor la distanță în Java este furnizarea unui cadru pentru a realiza o comunicare între obiecte prin intermediul metodelor, vizavi de localizarea lor. Asta înseamnă că un client trebuie să poată accesa un server din rețea ca și cum s-ar executa în același sistem.

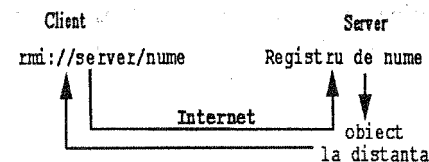
Pentru crearea unei clase care va fi accesibilă de la distanță, trebuie să definim și să implementăm o interfață care stabilește metodele disponibile (la distanță). Parametrii și valorile returnate de aceste metode pot fi de orice tip: transferul de date este rezolvat automat de fluxul de obiecte. Există un instrument disponibil sub JDK pentru compilarea claselor care pot fi invocate de la distanță, numit *rmic* (eng. *remote method invocation compiler*), care generează două fișiere de cod binar Java (cu extensia *.class*) importante:

- stub (buturugă, ciot)
- skeleton (schelet)

Stub-ul este o clasă care translatează automat apelurile metodelor la distanță în comunicații de rețea și trimitere de parametri. Skeleton-ul este clasa care staționează pe mașina virtuală de la distanță și care acceptă aceste conexiuni de rețea, pe care le translatează apoi în apeluri de metode actuale pe obiectul actual.



Pentru a putea fi utilizat obiectul de la distanță trebuie să fie înregistrat de un serviciu de nume care permite clienților să localizeze respectivul obiect. Clientul se conectează la un registru de nume și cere o referință la un serviciu înregistrat sub un nume dat (de exemplu, *Server RMI*). Registrul de nume întoarce apoi o referință la distanță la un obiect listat sub acest nume.



În arhitectura RMI, această referință include gazda unde obiectul la distanță rulează, portul la care ascultă și identificatorul RMI al obiectului intern. Programatorul primește un *stub* care implementează interfața cerută și translatează automat apelurile metodei în apeluri la distanță pentru obiectul real.

După ce clientul a obținut o referință la distanță, acesta poate proceda la apelul metodelor pentru obiectul de la distanță. Cadrul RMI are grijă de toate comunicațiile la nivel rețea. Clientul este capabil să realizeze apeluri de metode ale obiectului de la distanță ca și cum ar face apeluri la un obiect local. Cadrul RMI include chiar și suport pentru returnarea excepțiilor de la distanță și pentru colectarea „gunoiului” distribuit, așa încât obiectele de la distanță nu vor fi colectate atâta timp cât referințele de la distanță rămân.

Există impresia că se apelează direct de client obiectul de la distanță. Apelul este de fapt trimis *stub*-ului, care rezolvă toate detaliile de comunicare, transmițând apelul și parametrii către *skeleton*. Acesta face atunci un apel al metodei actuale pentru obiectul de la distanță. Valoarea returnată este în final trimisă înapoi de la *skeleton* la *stub*, care apoi returnează rezultatul clientului. Excepțiile sunt trimise înapoi, ca de altfel și rezultatele corecte.

Dacă un obiect de la distanță este setat corect, atunci toată această muncă este transparentă pentru obiectul de la distanță și pentru client. Nu toate metodele obiectului de la distanță pot fi apelate, ci doar acele metode declarate în interfața de la distanță. Cu aceste diferențe, cadrul RMI este un mecanism extrem de puternic, deoarece el permite dezvoltarea aplicațiilor de rețea ca și cum ar fi un apel de metodă obișnuit.

9.5.1. Obiectul de la distanță

Un obiect de la distanță este un obiect care a fost setat să accepte apeluri de metode de la un alt obiect care rulează într-o mașină virtuală Java la distanță. Aceasta necesită două etape:

- declararea unei interfețe care descrie metodele obiectului;
- implementarea acestei interfețe.

Această interfață trebuie să extindă interfața `java.rmi.Remote`, care este doar un indicator (eng. *flag*) ce identifică interfețele accesibile la distanță. Interfața `Remote` este superinterfața tuturor interfețelor de la distanță. O interfață la distanță descrie metodele pe care le conține un obiect la distanță. Acestea sunt doar metode care pot fi accesate prin RMI de un client. Interfața `Remote` este doar o interfață de marcare și, astfel, nu are metode. Toți parametrii și rezultatul întors al unei metode la distanță trebuie să fie serializabile de fluxul de obiecte.

Când un client RMI accesează un obiect la distanță, acesta va putea accesa obiectul doar prin una sau mai multe interfețe la distanță. Beneficiul este că implementarea este separată de interfața publică.

Exemplul 9.5.1. Declararea unei interfețe la distanță:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface InterfataLaDistanța extends Remote {
    public int metodaLaDistanța(String s) throws RemoteException;
}
```

În exemplul de mai sus, declarăm o interfață, care poate fi implementată de către orice obiect la distanță și care declară o singură metodă accesibilă la distanță, numită `metodaLaDistanța()`, cu un singur parametru de tip `String` și care întoarce un întreg. Toate metodele declarate într-o interfață la distanță trebuie să declare că pot arunca o excepție de tip `java.rmi.RemoteException`.

Implementarea obiectului de la distanță trebuie să importe pachetul `java.rmi.server`, să extindă `RemoteObject` (sau o subclasă a acestuia) și trebuie să implementeze fiecare dintre interfețele de la distanță pe care dorește să le implementeze. De obicei, se extinde subclasa `UnicastRemoteObject` care furnizează o implementare a comportărilor unui obiect obișnuit la distanță.

Exemplul 9.5.2. O implementare (minimală) a interfeței declarate în exemplul 9.5.1:

```
import java.rmi.*;
import java.rmi.server.*;

public class ImplementareLaDistanța extends UnicastRemoteObject
```

```
implements InterfataLaDistanța {
    /** constructor fara parametri */
    public ImplementareLaDistanța() throws RemoteException {}

    /** implementarea metodei din interfața */
    public int metodaLaDistanța(String s) throws RemoteException {
        /** returneaza lungimea sirului de caractere */
        return s.length;
    }
}
```

Clientul stub și obiectul de la distanță skeleton conțin tot codul implicat în detalii de serializare a apelului metodei la distanță. Pentru serializare se utilizează clasa `ObjectOutputStream`; trimerile se realizează prin conexiuni socket TCP/IP, iar pentru deserializare se recurge la clasa `ObjectInputStream`. Trimiterea obiectelor are loc întotdeauna doar prin valoare.

`RemoteException` este superclasa tuturor excepțiilor care pot apărea într-o execuție RMI. Această excepție este aruncată când eșuează un apel de metodă la distanță. Toate metodele dintr-o interfață la distanță trebuie să declare faptul că pot arunca acest tip de excepție.

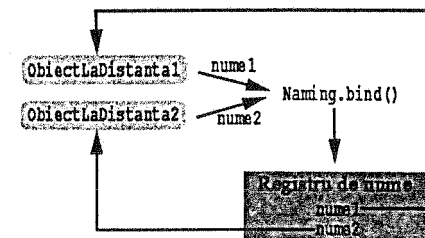
9.5.2. Registrul de nume

Clienții pot apela metode ale obiectelor la distanță doar dacă acestea au o referință la distanță către obiect. Cadrul RMI furnizează un registru de nume pentru obținerea acestor referințe.

Un registru de nume poate fi pornit manual prin comanda `rmiregistry` sau programatic cu ajutorul metodelor din clasa `LocateRegistry`.

Obiectele de la distanță pot fi înregistrate utilizând clasa `java.rmi.Naming` care folosește o schemă de numire asemănătoare cu cea de la URL-uri. Un registru de nume este el însuși un client simplu RMI, deci acesta trebuie să aibă acces la fișierele stub pentru toate clasele care sunt înregistrate; adică acestea trebuie să fie stocate într-un director care se regăsește în `CLASSPATH`.

Când un obiect se înregistrează cu registrul de nume, acesta specifică un nume sub care trebuie referit. Clieții pot utiliza clasa `java.rmi.Naming` pentru a obține o referință la distanță a obiectului din registru. Obiectul este simplu identificat ca un URL care conține numele gazdei pe care registrul de nume rulează și numele sub care obiectul este referit.



9.5.3. Exemplu de utilizare a apelurilor la distanță

În această secțiune prezentăm o aplicație client/server care permite clienților să afle data și ora de la server, utilizând apeluri de metode la distanță. Pașii creării aplicației sunt:

1. Definirea interfeței la distanță (este interfața prin care clienții de la distanță vor accesa serverul).
2. Implementarea interfeței de la distanță. Apelurile metodelor de la distanță de la client vor fi realizate prin intermediul acestei implementări. Acestea trebuie înregistrate într-un registru de nume pentru a putea fi localizate la distanță.
3. Generarea stub-ului și skeleton-ului utilizând comanda `rmic`.
4. Scrierea unui client care localizează serverul într-un registru de nume și apoi apelarea metodelor la distanță.
5. Pornirea registrului de nume folosind comanda `rmiregistry`. Serverul poate porni și el un registru.
6. Pornirea serverului.
7. Execuția clientului.

Pentru realizarea apelurilor metodelor aflate la distanță, vom urmări etapele mai sus menționate.

1. Declararea interfeței la distanță (`ServerDeDate`):

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface ServerDeDate extends Remote {
    /** metoda pentru aflarea datei */
    public Date obtineData() throws RemoteException;
}
```

2. Definirea clasei care implementează interfața de mai sus:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

public class ImplementareServerDeDate extends UnicastRemoteObject
    implements ServerDeDate {
    /** declararea unui constructor fara parametri */
    public ImplementareServerDeDate() throws RemoteException {}

    /** implementarea metodei pentru aflarea datei */
    public Date obtineData() {
        return new Date();
    }
}
```

```
public static void main(String [] args) throws Exception {
    /** obtinerea unei instante a clasei curente */
    ImplementareServerDeDate serverDeDate = new
        ImplementareServerDeDate();
    /** inregistrarea obiectului la distanta */
    Naming.bind("Date", serverDeDate);
}
}
```

3. Generarea stub-ului și skeleton-ului cu ajutorul comenzii:

```
rmic ImplementareServerDeDate
```

Se vor genera două clase: `ImplementareServerDeDate_Stub.class` și `ImplementareServerDeDate_Skel.class`.

4. Implementarea clasei corespunzătoare clientului:

```
import java.rmi.Naming;
import java.util.Date;

public class DataClient {
    public static void main(String args[]) throws Exception {
        if (args.length != 1)
            throw new IllegalArgumentException(
                "Sintaxa: ClientData <numeGazda>");
        // folosim metoda lookup() din clasa Naming
        // pentru colectarea unei referinte
        // a implementarii ServerDeDate. Un URL cu acest
        // camp protocol este folosit pentru numele
        // obiectului; partea cu rmi poate fi omisa deci
        // putem pune doar ca URL //server/obiect
        /** obtinerea unei referinte la un obiect aflat la distanta */
        ServerDeDate serverData = (ServerDeDate) Naming.lookup(
            "rmi://" + args[0] + "/Date");
        /** apelarea unei metode la distanta */
        Date data = serverData.obtineData();
        /** afisarea rezultatului */
        System.out.println(data);
    }
}
```

5. Pornirea registrului de pe mașina server utilizând comanda:

```
rmiregistry
```

Acest registru este acum accesibil clienților de la distanță pe un port standard (bine cunoscut). Comanda este executată din directorul unde sunt plasate fișierele stub.

6. Pornirea serverului se realizează astfel:

```
java ImplementareServerDeDate
```

7. Execuția clientului (pe mașina locală):

```
java DataClient localhost
```

sau (echivalent):

```
java ClientData 127.0.0.1
```

Continuăm cu explicații referitoare la execuția aplicației de mai sus. Clasa `Naming` se conectează la registrii care rulează pe gazda specificată și apoi caută obiectul care se potrivește cu numele indicat. Obiectul returnat de acest apel este o referință la distanță (de fapt, o instanță a unui `stub`). Astfel, trebuie să convertim rezultatul la interfața de la distanță așteptată, și nu numele clasei implementate la distanță. Doar metodele declarate în interfețele de la distanță sunt accesibile prin RMI.

Când s-a obținut o referință la distanță a obiectului, putem apela metode declarate în interfețele lor de la distanță. În acest caz, facem un apel la metoda `obtainData()` și afișăm la consolă răspunsul. Când realizăm acest apel de metodă, `stub`-ul local se conectează la `skeleton`-ul de la distanță, care apelează metoda obiectului de la distanță, obține rezultatul și prin intermediul fluxurilor se va trimite înapoi la `stub`.

9.5.4. Localizarea fișierelor RMI. Pachete necesare

Din punct de vedere al unei aplicații RMI, există cinci tipuri de fișiere `.class`: cu implementarea clientului, a serverului, interfața de la distanță, clasele `stub` și `skeleton`.

Pentru distribuirea unui client RMI trebuie să includem clasele de implementare a clientului, clasele de interfață la distanță și clasele `stub`.

Pentru distribuirea unui server RMI trebuie să includem clasele de implementare a serverului, interfețele de la distanță, clasele `stub` și clasele `skeleton`. Toate aceste clase sunt necesare în timpul rulării serverului și cu toate că `stub`-ul este executat doar de client, acesta este necesar când se exportă la distanță.

Anumite aplicații RMI nu separă așa de categoric partea de client și cea de server. Este posibil ca și serverul să poată executa anumite metode la distanță (nu numai clientul).

Principalele pachete Java care sunt utilizate de aplicațiile care invocă metode la distanță sunt:

- `java.rmi` - oferă suport pentru partea de client RMI; include interfețele prin care sunt accesate obiectele la distanță și un mecanism pentru localizarea serviciilor RMI pe o mașină la distanță;
- `java.rmi.server` - conține clase referitoare la partea de server RMI; include clase pentru lucrul cu servicii RMI, pentru rezolvarea cererilor TCP/IP și HTTP;
- `java.rmi.registry` - pune la dispoziție clase pentru lucru cu regiștri de nume RMI (creare, localizare și manipulare la distanță);

- `java.rmi.dgc` - conține clase referitoare la eliberarea memoriei (eng. *garbage collector*); serverele RMI păstrează automat un număr de referințe la distanță active pe care le servesc și le pot închide dacă nu mai sunt referite;
- `java.rmi.activation` - oferă suport pentru salvarea stărilor obiectelor la distanță atunci când sunt neutilizate și respectiv restaurarea acestora când apar cereri. Serverele RMI nu trebuie neapărat să ruleze mereu; ele pot fi activate de un „demon de activare” doar dacă sosește o cerere. Ele sunt ținute într-o stare pasivă serializată.

9.5.5. Clasa Naming

Clasa `Naming` face parte din pachetul `java.rmi` și oferă posibilitatea manipulării regiștrilor de nume RMI, localizarea și înregistrarea obiectelor. În interior, această clasă utilizează interfața `Registry` și clasa `LocateRegistry`, și furnizează metode statice și notații URL pentru adresarea obiectelor. De regulă, un client poate utiliza această clasă pentru localizarea unui obiect la distanță, iar un server, pentru înregistrare.

Un URL pentru RMI obișnuit are forma: `rmi://numeGazda:port/serviciu`. Acesta identifică obiectul la distanță, numit `serviciu`, memorat într-un registru de nume care rulează la portul specificat de pe gazda indicată. De exemplu, adresa `rmi://dil.infoiasi.ro:2500/java/exemple` se referă la un obiect la distanță înregistrat sub numele de `java/exemple` într-un registru de nume care ascultă la portul 2500 al mașinii `dil.infoiasi.ro`.

Toate metodele clasei `Naming` sunt statice și nu posedă constructori expliciți. Metodele acestei clase sunt descrise în tabelul următor:

Prototipul metodei	Descrierea metodei
<code>Remote lookup(String adresa) throws MalformedURLException, RemoteException, NotBoundException</code>	Se conectează la registru de nume specificat și extrage o referință a obiectului de la distanță înregistrat sub numele de serviciu indicat.
<code>void bind(String adresa, Remote obiect) throws MalformedURLException, RemoteException, AlreadyBoundException</code>	Stabilește numele serviciului (adresa) pentru obiectul specificat.
<code>void rebind(String adresa, Remote obiect) throws MalformedURLException, RemoteException</code>	Înlocuiește obiectul la care se referă serviciul, adresa cu obiectul specificat.
<code>void unbind(String adresa) throws RemoteException, NotBoundException, MalformedURLException</code>	Elimină serviciul specificat.
<code>String [] list(String adresa) throws MalformedURLException, RemoteException</code>	Returnează lista tuturor serviciilor disponibile la distanță înregistrate în registru de nume la adresa specificată.

Excepția `MalformedURLException` apare atunci când adresa RMI nu este validă, `RemoteException` dacă regiștri de nume nu pot fi consultați, `NotBoundException`...

dacă serviciul nu este încă legat (setat), iar `AlreadyBoundException` când se încearcă stabilirea unui nume de serviciu care deja există.

9.5.6. Clasa `LocateRegistry`

Clasa `LocateRegistry`, din pachetul `java.rmi.registry`, este utilă pentru controlul regiștrilor de nume. Aceasta furnizează metode pentru obținerea referințelor la distanță către regiștrii de nume.

Toate metodele acestei clase sunt statice și pot arunca excepția `RemoteException`. Acestea returnează obiecte care pot fi manipulate prin intermediul interfeței `Registry`. Metodele clasei `LocateRegistry` sunt:

Prototipul metodei	Descrierea metodei
<code>Registry getRegistry() throws RemoteException</code>	Returnează o referință la distanță pentru registrul de nume de pe mașina locală la portul implicit.
<code>Registry getRegistry(int port) throws RemoteException</code>	Întoarce o referință la distanță pentru registrul de nume de la portul specificat al mașinii locale.
<code>Registry getRegistry(String gazda) throws RemoteException</code>	Returnează o referință la distanță pentru registrul de nume aflat pe mașina gazdă specificată, la portul implicit.
<code>Registry getRegistry(String gazda, int port) throws RemoteException</code>	Întoarce o referință la distanță pentru registrul de nume de la portul și gazda indicate.
<code>Registry createRegistry(int port) throws RemoteException</code>	Creează și pornește un registru de nume pe mașina locală la portul specificat. Returnează o referință directă a registrului rezultat, nu o referință la distanță, cum returnează celelalte metode.

Exemplul 9.5.3. Căutarea unui registru de nume utilizând clasa `LocateRegistry`:

```
Registry registru = LocateRegistry.getRegistry("gazda", 1234);
registru.rebind("Serviciu", serviciu);
```

Crearea unui registru de nume local este simplă:

```
Registry registru = LocateRegistry.createRegistry(1234);
registru.bind("Serviciu", serviciu);
```

9.5.7. Interfața `Registry`

Această interfață din pachetul `java.rmi.registry` descrie partea publică a registrului de nume RMI. Acesta este implementat ca un obiect la distanță; clasa `LocateRegistry` va returna referințe la distanță pentru implementarea JDK a acestei interfețe.

Interfața `registry` declară o singură constantă statică:

- `int REGISTRY_PORT;`

care este numărul portului implicit unde ascultă registrul de nume (inițial, este valoarea 1099).

Spre deosebire de clasa `Naming`, când specificăm un nume de serviciu prin intermediul acestei interfețe, se poate specifica doar numele serviciului, nu și adresa registrului de nume. Metodele clasei `Naming` sunt de fapt implementate prin apeluri ale clasei `LocateRegistry`, urmate de apeluri la distanță cu ajutorul interfeței `Registry`.

Metodele interfeței `Registry` sunt următoarele:

Prototipul metodei	Descrierea metodei
<code>Remote lookup(String nume) throws RemoteException, NotBoundException, AccessException</code>	Returnează o referință la distanță către serviciul înregistrat în registrul de nume sub numele specificat.
<code>void bind(String nume, Remote object) throws RemoteException, AlreadyBoundException, AccessException</code>	Stocază o referință a obiectului de la distanță specificat în registrul sub numele nume.
<code>void rebind(String nume, Remote object) throws RemoteException, AccessException</code>	Modifică obiectul de la distanță cu cel specificat în registrul de nume, suprascriind orice intrare existentă.
<code>void unbind(String nume) throws RemoteException, NotBoundException, AccessException</code>	Șterge orice serviciu înregistrat sub numele nume din registru de nume.
<code>String [] list() throws RemoteException, AccessException</code>	Returnează lista completă a tuturor serviciilor disponibile din acest serviciu. Doar numele serviciului este returnat; nu este inclusă nici o informație despre adresă.

9.5.8. Clasa `RemoteObject`

Clasa `java.rmi.server.RemoteObject` este superclasa tuturor obiectelor la distanță. Aceasta suprascrie metodele `hashCode()`, `equals()` și `toString()` pentru a reflecta semanticele corecte ale obiectului la distanță. Clasa implementează interfața `Serializable` pentru implementarea corectă a serializării obiectelor de la distanță. Mai implementează și interfața `Remote`. Această clasă nu posedă alte metode importante în afară de cele mai sus menționate.

9.5.9. Obiecte la distanță ca parametri

Cu toate că RMI utilizează fluxuri de obiecte pentru trimiterea parametrilor către un apel de metodă la distanță, acesta, de fapt, folosește subclase specializate care suprascriu serializarea implicită pentru obiecte la distanță.

Când se trimite o implementare a unui obiect la distanță (a subclasei `RemoteObject`) ca parametru către un apel de metodă la distanță, se trimite, de fapt, o

referință la distanță către o implementare de obiect la distanță și nu o copie a acestuia (ca în cazul obiectelor locale). Așadar, când metoda de la distanță apelează metode ale unor obiecte pe care le primește, atunci au loc mai multe apeluri RMI pentru implementarea obiectului de la distanță original.

Consecința este că nu se poate trimite un obiect de la distanță ca parametru pentru un apel al unei metode la distanță și să ne așteptăm ca destinatarul să primească o instanță a clasei obiectului de la distanță actual. În schimb, destinatarul va primi o instanță a clasei stub care conține în interior o referință de la distanță către implementarea obiectului actual.

Exemplul 9.5.4. Transmiterea unui obiect la distanță ca parametru pentru o metodă aflată la distanță:

```
public class OImplementare extends UnicastRemoteObject implements
    InterfataNoastra {
    // ...
    public static void main(String [] args) throws Exception {
        OImplementare ob1 = new OImplementare();
        AltaDistanța ob2 = (AltaDistanța) Naming.lookup(
            "//server/ob2");
        ob2.apelare(ob1);
    }
}
```

În fragmentul de cod de mai sus am creat un obiect la distanță local, ob1, pe care îl trimitem ca parametru unei metode a altui obiect la distanță, ob2. După transferul de date RMI, metoda apelare() nu va primi un obiect de tip OImplementare, ci un obiect de tip OImplementare_Stub care implementează interfața InterfataNoastra și utilizează metodele definite de aceasta.

Deci, este greșit să definim o metodă la distanță care primește ca parametru un obiect de la distanță actual definit în clasa implementată. Se pot defini metode la distanță care primesc ca parametri fie obiecte locale, fie interfețe la distanță.

Sunt posibile următoarele declarații pentru metoda apelare():

- public void apelare(Object obiect) throws RemoteException;
- Signatura acestei metode este validă, deoarece se va transmite un obiect de tip OImplementare_Stub care este subclasă a lui Object;
- public void apelare(InterfataNoastra distanta) throws RemoteException;

Această variantă este validă, deoarece OImplementare_Stub implementează InterfataNoastra.

Următoarea declarație este nevalidă:

- public void apelare(OImplementare obiect) throws RemoteException;

Nu se obține eroare la compilare. Cu toate acestea, cadrul RMI va converti o instanță a clasei OImplementare într-o instanță pentru OImplementare_Stub în timpul serializării. Clasa stub nu este o subclasă a OImplementare_Stub și la execuție vom obține o excepție RMI.

9.5.10. Serializarea unui obiect la distanță

Când serializăm un obiect la distanță prin intermediul fluxurilor de obiecte locale, se va serializa un obiect la distanță valid. La deserializare, obiectul este automat restaurat. Utilizând acest mecanism, putem serializa un obiect la distanță pe disc și apoi să-l deserializăm fie cu aceeași mașină virtuală, fie cu ajutorul alteia. În urma acestui proces, va fi creat un obiect la distanță valid.

Exemplul 9.5.5. Serializarea unui obiect la distanță.

```
// ObjectOutputStream obiectDeIesire;
OImplementare ob1 = new OImplementare();
obiectDeIesire.writeObject(ob1);
```

Instanța lui OImplementare este serializată, utilizând procesul de serializare obișnuit.

Exemplul 9.5.6. Restaurarea unui obiect la distanță.

```
// ObjectInputStream obiectDeIntrare;
OImplementare ob2 = (OImplementare) obiectDeIesire.readObject();
```

Obținem, astfel, un duplicat al obiectului de la distanță original. Acesta este un obiect la distanță complet activat care poate fi înregistrat într-un registru de nume și manipulat exact ca o instanță originală.

9.5.11. Clasa RemoteServer

Clasa java.rmi.server.RemoteServer furnizează funcțiile de bază ale obiectelor la distanță care vor fi implementate ca servere (adică obiecte la distanță care vor deschide socket-uri pentru acceptarea de apeluri client).

Clasa RemoteServer extinde clasa RemoteObject și în plus posedă următoarele metode (toate sunt publice și statice):

Prototipul metodei	Descrierea metodei
String getClientHost() throws ServerNotActiveException	Când este apelată dintr-o implementare a unei metode la distanță, returnează adresa mașinii client, altfel aruncă o excepție de tip ServerNotActiveException.
void setLog(OutputStream iesire)	RMI furnizează posibilitatea de jurnalizare a tuturor apelurilor RMI. Această metodă stabilește care este fluxul pentru înregistrarea apelurilor.
PrintStream getLog()	Returnează fluxul pentru jurnalizare.

Exemplul 9.5.7. Apelul de mai jos permite înregistrarea tuturor apelurilor RMI în fișierul `rmi.log`.

```
RemoteServer.setLog(new FileOutputStream("rmi.log"));
```

Apelul de mai jos va închide fluxul pentru jurnalizare. Acesta nu închide fluxul original.

```
RemoteServer.setLog(null);
```

9.5.12. Clasa RemoteStub

Clasa `java.rmi.server.RemoteStub` este superclasa tuturor claselor `stub` de la distanță. Când se rulează `rmi`, fișierul `stub` produs este o subclasă a lui `RemoteStub`. Acesta implementează aceleași interfețe ca și clasa la distanță originală, dar furnizează toate metodele la distanță pentru implementarea obiectului de la distanță actual (prin conectare și comunicare cu `skeleton`-ul). Când se caută o referință la distanță într-un registru de nume, se primește, de fapt, o instanță a unei clase `stub`.

Clasa `RemoteStub` nu furnizează metode pentru uz particular.

9.5.13. Interfața Unreferenced

Interfața `Unreferenced` este utilă pentru obiecte la distanță care doresc să fie notificate de fiecare dată când nu există referințe la distanță înapoi către ele. Pentru implementarea colectorului de gunoi la distanță, cadrul RMI menține automat un contor al numărului de referințe care există către fiecare obiect la distanță. Această interfață face parte din pachetul `java.rmi.server`.

Există doar o singură metodă definită:

- `public void unreferenced()`

Aceasta va fi apelată automat de fiecare dată când nu mai sunt referințe la respectivul obiect (contorul pentru numărul de referințe ajunge la valoarea zero). Contorul de referință pentru obiectele aflate la distanță este independent de contorul de referință local. Acest mecanism este de obicei utilizat pentru dezactivare manuală a obiectelor, când acestea nu mai sunt referite.

O intrare într-un registru de nume este de fapt o referință la distanță către obiectul înregistrat, deci atât timp cât obiectul este memorat într-un registru de nume, acesta nu va fi nereferit.

Colectarea gunoierului distribuit implică notificarea periodică a referințelor la distanță ale unui obiect la distanță care sunt încă active. Perioada acestei notificări este chiar mare (implicit, 10 minute).

9.5.14. Clasa RMISocketFactory

Un obiect de tip `RMISocketFactory` este utilizat de cadrul RMI pentru obținerea socketului client sau server prin intermediul cărora au loc apeluri la distanță. Clasa `RMISocketFactory` este foarte importantă, deoarece permite stabilirea parametrilor implicați ai socketurilor utilizate în aplicațiile RMI.

Când cadrul RMI dorește să se conecteze la un obiect la distanță, acesta încearcă mai întâi să creeze o conexiune TCP directă către obiect. În acest caz, cererea poate fi de tipul următor:

```
raw://server:portObiectRMI
```

Din spatele unui *firewall*, o astfel de conexiune poate să nu fie permisă. Dacă se obține un eșec, atunci se încearcă o conexiune HTTP la un port la care obiectul de la distanță ascultă. Obiectele de la distanță furnizează un suport predefinit pentru transferul direct dintre date și transferul datelor prin intermediul protocolului HTTP. Astfel, cadrul RMI se va conecta la *proxy*-ul local (dacă acesta există) pentru procurarea obiectului de la distanță. Se va realiza o cerere de forma:

```
POST http://server:portObiectRMI/data . . .
```

Dacă *firewall*-ul sau *proxy*-ul permit doar traficul pe portul 80 al siturilor de la distanță, atunci se va realiza o conexiune cu un program CGI de pe serverul Web, care va directă cererea direct către obiectul de la distanță. În acest caz, cadrul RMI va trimite o cerere de forma:

```
POST http://server:80/cgi-bin/java-rmi.cgi/data . . .
```

Clasa `RMISocketFactory` este abstractă și deci nu îi putem crea instanțe. Metodele sale sunt descrise în tabelul de mai jos:

Prototipul metodei	Descrierea metodei
static void setSocketFactory (RMISocketFactory fabrica) throws IOException	Instalează o nouă „fabrică” de socketuri. Această clasă va fi responsabilă de crearea tuturor socketurilor viitoare în această mașină virtuală.
static RMISocketFactory getSocketFactory ()	Returnează fabrica de socketuri curentă dacă există sau null în caz contrar (când este utilizată cea implicită).
static RMISocketFactory getDefaultSocketFactory ()	Întoarce fabrica de socketuri implicită.
static void setFailureHandler (RMIFailureHandler rezolvitorEsec)	Stabilește un rezolvitor de eșec RMI. Acesta este apelat atunci când crearea unui socket RMI se încheie cu eșec.
static RMIFailureHandler getFailureHandler ()	Returnează rezolvitorul de eșec curent.
abstract Socket createSocket (String gazda, int port) throws IOException	Trebuie să returneze un nou Socket conectat la gazda și portul specificat. Orice excepție poate fi aruncată și trimisă mai departe sau poate fi rezolvată intern.
abstract ServerSocket createServerSocket (int port) throws IOException	Trebuie să returneze un nou ServerSocket care ascultă la portul specificat. Excepțiile pot fi tratate sau trimise mai departe.

Se poate stabili o singură dată fabrica de socketuri; metoda `setSocketFactory()` va arunca o excepție `IOException`, dacă este deja definită o fabrică. Această metodă este controlată și de managerul de securitate, deci codurile nesigure nu vor putea înregistra o fabrică socket.

9.5.15. Interfața `RMIFailureHandler`

Această interfață descrie un rezolvitor de eșec RMI. Ea încearcă să rezolve cauza unei excepții apărute la crearea unui socket RMI. Dacă se reușește rezolvarea cazului apărut, cadrul RMI poate reîncerca o nouă conectare.

Există o singură metodă declarată în această interfață:

- `boolean failure(Exception ex);`
care este apelată când crearea socketului RMI a eșuat, iar ca parametru este trimisă excepția apărută. O implementare ar trebui să încerce să rezolve problema și să returneze `true`, pentru a se mai încerca crearea socketului de către cadrul RMI.

9.5.16. Exemplu de aplicație RMI

Vom prezenta o implementare bazată pe RMI a unui sistem de discuții (eng. *chat*) de tip client/server. Serverul este dat de clasa `ImplementareServerChatRMI`, iar o instanță a acesteia este un obiect la distanță care menține o listă internă a tuturor mesajelor primite de la clienți. Clasa pentru server implementează interfața `ServerChatRMI`, prin care clienții pot adăuga noi mesaje și pot interoga lista curentă de mesaje.

Cliantul, `ClientChatRMI`, este o aplicație simplă; acesta trimite mesaje către server într-o manieră simplă și posedă un fir de execuție care periodic interoghează serverul pentru o copie actualizată a sesiunii chat.

Interfața `ServerChatRMI` stabilește metodele care vor fi accesibile la distanță:

```
import java.rmi.*;

/** declararea interfetei pentru obiectul de la distanta */
public interface ServerChatRMI extends Remote {
    /** Stabilirea numelui pentru registru */
    public static final String NUME_REGISTRU = "ServerChat";

    /** Definirea prototipului metodei pentru preluarea mesajelor noi */
    public abstract String[] obtineMesaje(int index)
        throws RemoteException;

    /** Definirea prototipului metodei pentru adaugarea unui mesaj */
    public abstract void adaugaMesaj(String mesaj)
        throws RemoteException;
}
```

Codul clasei `ImplementareServerChatRMI` este următorul:

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.util.*;

public class ImplementareServerChatRMI extends UnicastRemoteObject
    implements ServerChatRMI {
    /** Declararea datelor membre */
    protected Vector mesaje; //tabloul de mesaje

    /** Declararea constructorului fara parametri */
    public ImplementareServerChatRMI() throws RemoteException {
        /** Crearea vectorului pentru mesaje */
        mesaje = new Vector();
    }

    /** Implementarea metodelor din interfata ServerChatRMI */
    /** Obținerea mesajelor noi */
    public String[] obtineMesaje(int index) {
        /** Aflarea numarului total de mesaje */
        int dimensiune = mesaje.size();
        /** Crearea tabloului pentru mesajele noi */
        String[] actualizat = new String[dimensiune - index];
        /** Preluarea mesajelor noi */
        for (int i = 0; i < dimensiune - index; ++ i)
            actualizat[i] = (String) mesaje.elementAt(index + i);
        /** Returnarea rezultatului */
        return actualizat;
    }

    /** Adaugarea unui mesaj */
    public void adaugaMesaj(String mesaj) {
        /** adaugarea mesajului in vectorul de mesaje */
        mesaje.addElement(mesaj);
    }

    /** Metoda principala */
    public static void main(String [] args) throws
        RemoteException {
        /** Obținerea unei instanțe a clasei pentru server */
        ImplementareServerChatRMI serverChat = new
            ImplementareServerChatRMI();
        /** înregistrarea obiectului la distanta */
    }
```

```

Registry registru = LocateRegistry.getRegistry();
registru.rebind(NUME_REGISTRU, serverChat);
}
}

```

Pentru a putea executa serverul, trebuie compilat programul de mai sus (cu ajutorul comenzii `rmic ImplementareServerChatRMI`), rezultând fișierele *stub* și *skeleton*, apoi să pornim registrul de nume (utilizând comanda `rmiregistry`). Pornirea serverului se realizează cu ajutorul comenzii: `java ImplementareServerChatRMI`.

Aplicația client este următoarea:

```

import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.rmi.registry.*;

public class ClientChatRMI implements Runnable, ActionListener {
    /** definirea datelor membre */
    protected static final int PAUZA_ACTUALIZARE = 100;
    protected String gazda;
    protected Frame cadru;
    protected TextField intrare;
    protected TextArea iesire;
    protected ServerChatRMI server;
    protected Thread actualizator;

    /** definirea constructorului cu un parametru de tip String */
    public ClientChatRMI (String gazda) {
        this.gazda = gazda;
        /** crearea unei ferestre */
        cadru = new Frame("ClientChatRMI[" + gazda + "]");
        /** crearea unui element textarea pentru afisarea mesajelor */
        cadru.add(iesire = new TextArea(), "Center");
        iesire.setEditable(false);
        /** crearea unui camp text pentru inserarea mesajelor noi */
        cadru.add(intrare = new TextField(), "South");
        /** stabilirea ascultatorilor */
        intrare.addActionListener(this);
        // urmeaza declaratia unei clase interioare
        cadru.addWindowListener(new WindowAdapter() {
            public void windowOpened(WindowEvent ev) {
                intrare.requestFocus();
            }
            public void windowClosing(WindowEvent ev) {
                opreste();
            }
        });
    }
}

```

```

        System.exit(0);
    }
}); // sfarsitul apelului metodei addWindowListener
cadru.pack();
} // sfarsitul constructorului ClientChatRMI

/** Metoda pentru pornirea chat-ului */
public synchronized void porneste() throws RemoteException,
    NotBoundException {
    if (actualizator == null) {
        /** obtinerea registrului */
        Registry registru = LocateRegistry.getRegistry(gazda);
        /** obtinerea unei instante la obiectul la distanta */
        server = (ServerChatRMI) registru.lookup(
            ServerChatRMI.NUME_REGISTRU);
        /** crearea unui fir de executie */
        actualizator = new Thread(this);
        actualizator.start();
        /** vizualizarea ferestrei */
        cadru.setVisible(true);
    }
}

/** metode pentru oprirea clientului */
public synchronized void opreste() {
    /** intrerupem activitatea firului de executie */
    if (actualizator != null) {
        actualizator.interrupt();
        actualizator = null;
        server = null;
    }
    /** ascundem fereastra */
    cadru.setVisible(false);
}

/** executia firului de executie */
public void run() {
    try {
        /** incepem cu primul mesaj */
        int index = 0;
        /** citim mesaje atat timp cat nu este intrerupt firul
            de executie */
        while (!Thread.interrupted()) {
            /** obtinem mesajele noi */
            String [] mesaje = server.obtineMesaje(index);

```

```

    /** afisam mesajele in campul de tip textarea */
    int n = mesaje.length;
    for (int i = 0; i < n; ++ i)
        iesire.append(mesaje[i] + "\n");
    index += n;
    /** firul de executie va astepta un timp
        pana la preluarea altor mesaje */
    Thread.sleep(PAUZA_ACTUALIZARE);
}

/** tratarea exceptiilor */
catch (InterruptedException ie) {}
catch (RemoteException ex) {
    intrare.setVisible(false);
    cadru.validate();
    ex.printStackTrace();
}

} // sfarsitul metodei run()

/** metoda se apeleaza la apasarea tastei ENTER in campul
text */
public void actionPerformed(ActionEvent ev) {
    try {
        ServerChatRMI server = this.server;
        /** daca suntem conectati trimitem mesajul */
        if (server != null)
            server.adaugaMesaj(ev.getActionCommand());
        /** stergem mesajul trimis */
        intrare.setText("");
    } /** Tratarea exceptiilor */
    catch (RemoteException ex) {
        Thread tmp = actualizator;
        actualizator = null;
        if (tmp != null)
            tmp.interrupt();
        intrare.setVisible(false);
        cadru.validate();
        ex.printStackTrace();
    }
} // de la public void actionPerformed(ActionEvent ev)

/** metoda principala */
public static void main(String [] args)
    throws RemoteException, NotBoundException {
    /** primul parametru va fi numele masinii server */

```

```

    if (args.length != 1)
        throw new IllegalArgumentException
            ("Sintaxa: ClientChatRMI <gazda>");
    /** cream o instanta a clientului */
    ClientChatRMI clientChat = new ClientChatRMI(args[0]);
    /** pornim chat-ul */
    clientChat.pornește();
}
}

```

Compilarea și execuția se realizează la fel ca pentru orice aplicație Java de sine stătătoare (obișnuită).

Rămâne în sarcina utilizatorilor de a stabili un nume pentru fiecare utilizator, pentru a putea ști cine a trimis mesajele. La anumite intervale de timp va trebui să eliminăm mesajele vechi pentru a nu solicita prea multe resurse.

9.6. Concluzii

Prezentul capitol a debutat cu câteva elemente introductive care au stabilit termenii de bază vehiculați în programarea rețelor de calculatoare (adrese IP, porturi, socketuri, conexiuni, datagrame etc.). Apoi s-au prezentat diferențele dintre conexiuni și datagrame. Conexiunile sunt asemănătoare apelurilor telefonice (simultan, ambii parteneri de comunicare trebuie să fie disponibili), iar datagramele, sistemului de poștă obișnuită. Exemplele prezentate evidențiază ușurința lucrului cu socketuri atât pentru conexiuni, cât și pentru datagrame.

Clasa URL permite atât specificarea URL-urilor, cât și obținerea într-o manieră facilă a conținutului resurselor referite de acestea. Această facilitate reduce timpul de acces al resursei, precum și codul scris de programator.

O modalitate de programare a rețelor de calculatoare o reprezintă apelurile metodelor aflate la distanță (pe alte mașini din rețea). De data aceasta, comunicarea parametrilor și returnarea rezultatelor este realizată automat, iar lucrul cu obiectele aflate la distanță este similar cu cele locale. Obiectele aflate la distanță se pot serializa ca și obiectele obișnuite, la restaurarea acestora neexistând probleme.

La compilarea obiectelor aflate la distanță rezultă două fișiere de tip .class. Pentru a fi accesibil la distanță, un obiect trebuie să fie înregistrat la un serviciu de nume.

Atât invocarea metodelor la distanță, cât și socketurile permit realizarea de aplicații distribuite, execuția nemaifiind pe un singur calculator, ci pe mai multe mașini din rețea.

9.7. Test grilă

Întrebarea 9.7.1. Care dintre următoarele afirmații sunt false:

- a) Comunicarea prin datagrame nu asigură primirea corectă (nedeteriorată) a mesajelor.
- b) Protocolul HTTP este orientat conexiune.
- c) La un port pot asculta mai multe servere TCP.
- d) Doar un server TCP și unul UDP pot asculta la același port.

Întrebarea 9.7.2. Care dintre afirmațiile următoare sunt adevărate:

- a) O aplicație server poate asculta la mai multe porturi.
- b) Un socket poate să asculte la un singur port.
- c) O mașină din rețea poate avea mai multe adrese IP.
- d) O mașină din rețea poate avea un singur nume (domeniu).

Întrebarea 9.7.3. Apelurile de metode la distanță se realizează:

- a) prin conexiuni TCP;
- b) prin datagrame;
- c) prin intermediul protocolului HTTP;
- d) cu ajutorul *firewall*-urilor.

Întrebarea 9.7.4. Denumirea de URL vine de la:

- a) Universal Resource Location.
- b) Universal Resource Locator.
- c) Uniform Resource Location.
- d) Uniform Resource Locator.

Întrebarea 9.7.5. Pentru a apela metodele unui obiect la distanță trebuie să:

- a) pornim un registru de nume;
- b) creăm un socket pentru a stabili o conexiune cu serverul RMI;
- c) realizăm o cerere pentru respectivul obiect prin intermediul socketului creat anterior;
- d) obținem o instanță a obiectului la distanță.

Întrebarea 9.7.6. Care dintre afirmațiile următoare referitoare la trimiterea obiectelor la distanță ca parametri ai metodelor la distanță sunt adevărate?

- a) Nu se pot trimite.
- b) Se trimite o copie a lor.
- c) Se trimite o referință la distanță.
- d) Se trimite obiectul *stub* corespunzător.
- e) Se trimite obiectul *skeleton* corespunzător.

Întrebarea 9.7.7. Serializarea obiectelor la distanță se poate realiza:

- a) ca la obiectele locale;
- b) ca la obiectele locale, numai că la restaurare vor apărea probleme;

- c) într-un mod specific, RMI implementând interfața *RMISerializable*;
- d) nu se pot serializa obiectele la distanță.

9.8. Exerciții propuse spre implementare

Exercițiul 9.8.1. (Calculator) Scrieți o aplicație client/server în care clientul trimite o expresie aritmetică la server și serverul întoarce valoarea acesteia care se afișează la situl client.

Exercițiul 9.8.2. Să se implementeze un obiect la distanță cu o metodă care primește adresa de IP sau numele domeniului unei mașini din rețea și returnează clasa din care face parte (din cele cinci existente: A, B, C, D, E). Pentru stabilirea clasei din care face parte va trebui să cercetăm primul număr din adresa IP. Dacă acesta are primul bit 0, atunci este de clasă A, dacă primii 2 biți sunt 10, atunci este de clasă B, pentru 110 avem clasa C, 1110 pentru D și 11110 pentru E.

Exercițiul 9.8.3. Scrieți o aplicație client/server bazată pe RMI a cărei interfață conține două metode: una pentru inițializarea unui obiect de tip *Vector* și a doua pentru returnarea sumei elementelor vectorului.

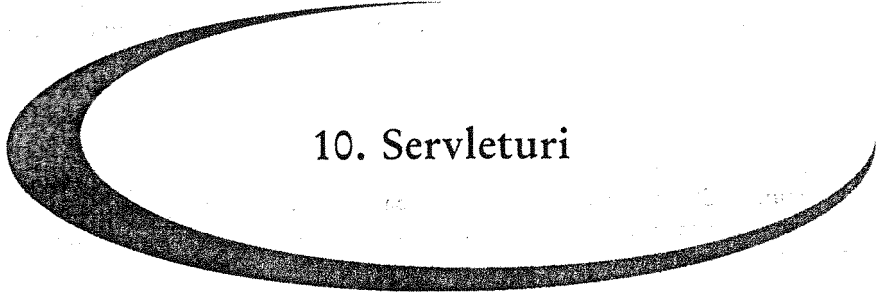
Exercițiul 9.8.4. Să se creeze o aplicație care permite specificarea adresei unui sit Web și verifică dacă acesta are „legături moarte” (legături la pagini inexistente).

Exercițiul 9.8.5. Să se implementeze un motor de căutare pentru situri Web. Aplicația va avea ca intrare adresa unui sit Web și o secvență de cuvinte, iar ca ieșire lista tuturor paginilor din respectivul sit care conțin respectiva secvență și contextul în care apare. Eventual, se va da posibilitatea de a realiza căutări doar în cadrul paginilor din ultimul rezultat obținut.

9.9. Proiecte propuse spre implementare

Proiectul 9.9.1. Scrieți o aplicație client/server care să simuleze un sistem (grafic) de chat. Clientul trimite un mesaj la server și serverul trimite la toți utilizatorii (clienții) acelui chat mesajul primit. Se va da posibilitatea de a vizualiza utilizatorii conectați, de a trimite mesaje individuale sau la grupuri de utilizatori, de a crea și administra „camere” de discuții, posibilitatea de a restricționa accesul persoanelor cu un comportament inadecvat.

Proiectul 9.9.2. Să se conceapă un sistem de tip Napster, în care fiecare utilizator pune la dispoziție o serie de fișiere (audio, video, componente software, programe etc.). Cu ajutorul unui serviciu de căutare se poate depista dacă există fișierele dorite de un anumit utilizator și persoanele care oferă respectivele fișiere. Se va da posibilitatea de a copia fișiere de la distanță (de la alți utilizatori).



10. Servleturi

10.1. Cuvinte cheie

- servlet, CGI
- server Web, Tomcat

10.2. CGI (Common Gateway Interface)

Standard pentru interacțiunea clienților Web cu serverele Web, *Common Gateway Interface* se află în prezent la versiunea 1.1. Un program executabil sau script CGI se execută de către serverul Web, fie în mod explicit (apelat din cadrul paginii printr-o directivă specială), fie la preluarea informațiilor aflate în cadrul câmpurilor unui formular interactiv sau coordonatelor unei zone senzitive (*image map*). Acest standard conferă interactivitate paginilor Web, documentele HTML putând astfel să-și modifice în mod dinamic conținutul și să permită prelucrări sofisticate de date.

Programele CGI pot fi scrise în orice limbaj, fiind interpretate (cazul Perl, Tcl, REXX, scripturi shell Unix) sau compilate (C, C++, Delphi). Regulile care trebuie respectate la conceperea unui program CGI sunt:

- programul scrie datele la ieșirea standard (stdout);
- programul generează antete care permit serverului Web să interpreteze corect ieșirea scriptului (folosindu-se specificațiile protocolului HTTP);
- datele de intrare se iau de la intrarea standard (stdin) sau din variabilele de mediu.

Cele mai multe script-uri CGI sunt concepute pentru a procesa datele introduse în formulare. Un formular se definește în HTML folosindu-se tag-uri specifice pentru afișarea conținutului și introducerea datelor de către client (vezi specificațiile pentru HTML 4.01 sau XHTML 1.0 ale Consorțiului Web), iar script-ul CGI, executat de server, va prelua conținutul acelui formular și-l va prelucra, returnând, eventual, rezultatele.

Antetul trimis serverului de către programul CGI va respecta specificațiile *MIME* (*Multipurpose Internet Mail Extensions*), conținând, de exemplu, directiva Content-type pentru a indica tipul de document generat. De exemplu, Content-type: text/html va indica faptul că rezultatul procesării datelor este un document HTML, Content-type: image/jpeg este o imagine în format JPEG, iar Content-type: text/plain este text obișnuit.

Tipurile MIME primare sunt sintetizate în următorul tabel:

Tip	Descriere	Exemple
application	definește aplicațiile client	application/postscript application/octet-stream
audio	definește formatele audio	audio/basic audio/aiff audio/mid
image	definește formatele grafice	image/gif image/jpeg image/png
text	definește formatele text	text/plain text/html text/xml text/css

Tip	Descriere	Exemple
video	definește formatele video	video/mpeg video/quicktime
multipart	utilizat pentru transmisia informațiilor compuse	multipart/mixed multipart/alternative
message	utilizat pentru transmisia mesajelor de poștă electronică	message/partial message/rfc822

Tipurile sau subtipurile MIME nestandardizate încă vor fi prefixate de caracterele „x-”. Astfel, putem întâlni application/x-cdf (definiție de canal de date activ), image/x-targa (format grafic Targa) sau x-world/x-vrml (lume virtuală modelată în VRML; acest tip MIME este acum înlocuit cu model/vrml).

Programului CGI i se vor transmite în variabile de mediu sau de la intrarea standard informații referitoare la datele de procesat sau la clientul care a inițiat cererea (de exemplu, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, QUERY_STRING, REMOTE_HOST etc.) în funcție de metoda de transfer utilizată.

10.2.1. Variabilele de mediu

Oricare program CGI va putea dispune de valorile următoarelor variabile de mediu:

- Variabile independente de metoda cererii și setate pentru toate cererile HTTP:
 - SERVER_SOFTWARE furnizează numele și versiunea serverului Web care procesează cererea HTTP și rulează scriptul CGI.
 - SERVER_NAME dă numele de host, alias-ul DNS sau adresa IP a mașinii pe care rulează serverul Web.
 - GATEWAY_INTERFACE stochează versiunea specificației CGI folosite (CGI/1.0 ori CGI/1.1 în prezent, CGI/1.2 în viitorul apropiat).
- Variabile de mediu specifice cererilor care vor fi transmise spre programul CGI:
 - SERVER_PROTOCOL indică numele și versiunea protocolului de transmitere a datelor (e.g. HTTP/1.0).
 - SERVER_PORT furnizează portul asociat serverului care va procesa cererea (de obicei, portul 80).
 - REQUEST_METHOD specifică metoda prin care va fi formulată cererea: GET, POST, PUT etc.
 - SCRIPT_NAME desemnează calea virtuală până la scriptul care va fi executat.
 - QUERY_STRING conține toate informațiile de după caracterul „?” din URL-ul care referă scriptul CGI prin metoda GET, informații care vor trebui decodificate de acel script.
 - REMOTE_HOST indică host-ul care a formulat cererea (calculatorul care a apelat scriptul CGI prin intermediul unei pagini Web).
 - REMOTE_ADDR furnizează adresa IP a clientului care a formulat cererea, asociată adresei simbolice date de REMOTE_HOST.
 - AUTH_TYPE dă metoda de autentificare utilizată să valideze utilizatorul, dacă serverul suportă autentificarea utilizatorilor și scriptul este protejat.

- `REMOTE_USER` indică numele utilizatorului, dacă serverul suportă autentificarea utilizatorilor.
- `CONTENT_TYPE` indică tipul conținutului datelor vehiculate, pentru cererile care au atașate informații suplimentare, precum PUT sau POST (de exemplu, pentru un formular transmis prin metoda POST, normal ar trebui ca variabila `CONTENT_TYPE` să aibă valoarea `application/x-www-form-urlencoded`).
- `CONTENT_LENGTH` conține numărul de octeți ai datelor trimise de client.

Suplimentar, liniile antet (dacă există) recepționate de la client prin intermediul protocolului HTTP vor fi plasate în variabile de mediu având numele prefixate de `HTTP_`. Ca exemple de astfel de variabile, pot fi menționate:

- `HTTP_ACCEPT` indică tipurile MIME pe care le acceptă clientul, fiecare tip fiind despărțit de virgulă, așa cum specifică protocolul HTTP: *tip/subtip, tip/subtip*.
- `HTTP_USER_AGENT` furnizează numele și versiunea navigatorului Web care a formulat cererea, în formatul general: *software/versiune biblioteca/versiune*.

Exemplul 10.2.1. Următorul script Perl va afișa toate variabilele de mediu la care are acces prin intermediul interfeței CGI:

```
#!/usr/bin/perl
# Afiseaza mediul
print "Content-type: text/plain\n\n";
print "Environment:\n\n";
foreach $key (keys %ENV) {
    printf "%16s = %s\n", $key, $ENV{$key};
}
```

Variabila predefinită `ENV` este un tablou asociativ conținând perechile (*nume de variabilă de mediu, valoarea variabilei*).

Exemplul 10.2.2. Un script CGI similar, elaborat în *bash* (*Bourne Again Shell*), este:

```
#!/bin/bash
# Afisează mediul
echo "Content-type: text/plain"
echo
set
```

Un posibil rezultat al execuției scriptului de mai sus poate fi:

```
HTTP_ACCEPT_LANGUAGE=en
SERVER_SIGNATURE=<ADDRESS>Apache/1.3.12 Server at endirra.ro
Port 80</ADDRESS>
SCRIPT_FILENAME=/home/httpd/cgi-bin/env.cgi
HTTP_ACCEPT_CHARSET=iso-8859-1,*,utf-8
```

```
HTTP_PRAGMA=no-cache
SERVER_NAME=endirra.ro
HTTP_CONNECTION=Keep-Alive
REMOTE_HOST=thor.infoiasi.ro
REMOTE_ADDR=193.231.30.225
REQUEST_URI=/cgi-bin/env.cgi
HTTP_HOST=endirra
REMOTE_PORT=1041
REQUEST_METHOD=GET
GATEWAY_INTERFACE=CGI/1.1
QUERY_STRING=
SERVER_ADDR=127.0.0.1
SERVER_SOFTWARE=Apache/1.3.12 (Unix)
(Red Hat/Linux) PHP/4.0.2 mod_perl/1.2.1
SERVER_PROTOCOL=HTTP/1.0
SERVER_PORT=80
DOCUMENT_ROOT=/home/httpd/html
HTTP_USER_AGENT=Mozilla/4.75 [en] (X11; U; Linux 2.2.14-5.0 i686)
HTTP_ACCEPT=image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, image/png, */*
SCRIPT_NAME=/cgi-bin/env.cgi
SERVER_ADMIN=root@endirra.ro
HTTP_ACCEPT_ENCODING=gzip
```

10.2.2. Apelarea programelor CGI din formularele HTML

De cele mai multe ori, un script CGI va fi invocat din cadrul unui formular HTML la apăsarea butonului de trimitere a datelor către server (butonul de *submit*).

Exemplul 10.2.3. Un utilizator introduce prin intermediul unui formular două numere întregi, iar programul CGI va genera o pagină Web conținând suma lor. Formularul XHTML va fi:

```
<form action="http://www.infoiasi.ro/cgi-bin/suma.cgi"
method="GET">
  <p>Va rugam, introduceti doua numere:</p>
  <input name="nr1" size="5" />
  <input name="nr2" size="5" />
  <p>
    <input type="submit" value="Afla suma" /></p>
</form>
```

De exemplu, introducând numerele 7 și 4 și acționând butonul etichetat "Afla suma", se va primi o pagină Web (document HTML) care va afișa textul "Suma dintre 7 și 4 este 11" (aproape întotdeauna, $7 + 4 = 11$).

Presupunând că în cele două câmpuri ale formularului (purtaînd numele nr1 și respectiv nr2) am introdus valorile 7, respectiv 4 și am apăsat butonul submit, navigatorul va trimite, prin intermediul protocolului HTTP, o cerere către serverul Web aflat la adresa dată de URL-ul `http://www.infoiasi.ro` (adresa este preluată din valoarea atributului `action` din `<form>`). Desigur, în loc de o adresă absolută, putea fi specificată o cale relativă, însemnând faptul că se utilizează serverul pe care se găsește pagina conținând formularul.

Atunci când se trimite cererea HTTP, navigatorul construiește un URL având ca sufix informații despre datele introduse în câmpurile formularului, în cazul nostru `http://www.infoiasi.ro/cgi-bin/suma.cgi?nr1=7&nr2=4`, folosindu-se o codificare specială. Pentru fiecare câmp al formularului, se va genera o pereche *nume_câmp=valoare* delimitată de caracterul „&”, iar caracterele speciale (ca de exemplu slash-ul sau apostroful) vor fi înlocuite de codul lor numeric, în hexazecimal, precedat de caracterul procent („%”). Spațiile vor fi substituite de semnul plus („+”).

Serverul spre care cererea a fost expediată (în cazul sitului `www.infoiasi.ro`, un server *Apache*) va procesa datele recepționate conform regulilor proprii. Tipic, configurația serverului definește unde sunt stocate, în cadrul sistemului de fișiere, directoarele și fișierele CGI. De cele mai multe ori, serverul Web *Apache* rulează pe mașină sub auspiciu de utilizator fictiv (ca *nobody* sau, mai nou, *apache* – din rațiuni de securitate), fișierele sale fiind stocate implicit în directorul `/home/httpd/` sau, mai nou, în `/var/www/html/`, în cazul unui sistem Unix, iar pentru sistemul Windows în `C:\apache\htdocs`. Aici, alături de directorul `html` unde se memorează documentele HTML ale unui sit Web, se află și directorul `cgi-bin`, unde ar trebui să fie stocate toate fișierele CGI apelate din cadrul paginilor Web. Așadar, `http://www.infoiasi.ro/cgi-bin/suma.cgi` va însemna pentru serverul Web de la adresa `www.infoiasi.ro` următoarea acțiune: „*invocă programul suma.cgi aflat la /home/httpd/cgi-bin*”, dacă directorul pentru documentele Web este `/home/httpd`.

Acțiunea de *invocare* va avea o semantică diferită în funcție de programul CGI conceput. Pentru un script Perl, serverul va invoca un interpretor Perl (în cazul *Apache*, un modul special *mod_perl*). Pentru un program executabil (compilat, de pildă, într-unul din limbajele C sau C++, după cum vom vedea mai jos), serverul va lansa programul ca un proces separat.

10.2.3. Procesarea datelor din formularele HTML prin metoda GET

Pentru formularele care utilizează metoda GET (metoda implicită), specificațiile CGI indică faptul că datele furnizate scriptului vor fi disponibile într-o variabilă de mediu denumită `QUERY_STRING`.

Desigur, în funcție de limbajul de programare ales pentru conceperea scriptului CGI, datele vor fi accesate via `QUERY_STRING` și prelucrate. În limbajul C, vom putea folosi funcția `getenv()` aflată în biblioteca standard `stdlib.h`. Tot ce mai rămâne de făcut este să decodificăm valorile din variabila `QUERY_STRING`.

Exemplul 10.2.4. Scriptul CGI, în mod uzual, va genera date de ieșire care vor fi trimise navigatorului Web de către server, utilizându-se ieșirea standard (*stdout*):

```
#include <stdio.h>
#include <stdlib.h>

#define CONTENT "Content-type: text/html\n\n"

int main(void)
{
    char *data;
    long int nr1, nr2;

    /* obligatoriu, trimite tipul de date (în format MIME) */
    printf(CONTENT);
    /* antetul HTML */
    printf("<html>\n<head><title>Rezultat</title></head>\n");
    /* corpul documentului */
    printf("<body bgcolor=\"white\" text=\"black\">\n");
    /* proceseaza datele */
    data = getenv("QUERY_STRING");
    if (data == NULL)
        printf("<p>Eroare la procesarea datelor!</p>\n");
    else
        if (sscanf(data, "nr1=%ld&nr2=%ld", &nr1, &nr2) != 2)
            printf("<p>Eroare! Datele trebuie sa fie numerice!</p>\n");
        else
            printf("<p>Suma dintre %ld si %ld este %ld.</p>\n",
                    nr1, nr2, nr1 + nr2);
    /* terminare */
    printf("</body>\n</html>\n");
    return 0;
}
```

Am utilizat specificatorul de format `%ld` pentru că datele numerice sunt de tip `long int`. Din cauza faptului că avem un format fix al datelor recepționate, am folosit `sscanf()`, dar pentru decodificări mai complexe trebuie să scriem o funcție specială (vezi mai jos).

Executabilul, după compilare, va trebui plasat (în cele mai multe situații) în directorul `cgi-bin` al serverului Web. Pentru a putea fi accesat, va fi necesar să posede drepturi de execuție pentru serverul Web (de obicei, pentru utilizatorul special *nobody* sau *apache* în Unix).

După instalarea scriptului, orice utilizator din Internet (dacă nu sunt impuse restricții de către administratorul Web) îl va putea accesa și executa prin intermediul

unui formular HTML sau explicit. Astfel, scriptul va trebui să fie capabil să proceseze orice tip de dată și să semnaleze posibilele erori.

Însă metoda GET are câteva neajunsuri: poate trimite un număr limitat de caractere (octeți) prin intermediul variabilelor (de regulă 255 octeți), iar câmpurile de parolă vor fi vizibile în URL, ceea ce oferă o securitate redusă. Totuși, această metodă este frecvent utilizată, mai ales de motoarele de căutare.

10.2.4. Procesarea unui formular prin metoda POST

Un program CGI are acces la întreg mediul sistemului de operare pe care rulează. Metoda POST este utilă în situațiile în care avem de trimis script-ului CGI spre prelucrare un volum mai mare de date (de exemplu, conținutul unei scrisori introdus prin intermediul unui <textarea> ori al unui fișier, prin acțiunea de *upload*) sau informații confidențiale (e.g. parole) care nu trebuie să apară în componența URI-ului transmis serverului Web.

Pentru formularele utilizând metoda POST, datele trimise scriptului vor putea fi accesate de la intrarea standard (*stdin*), iar lungimea, în octeți, a datelor trimise va fi disponibilă în variabila de mediu *CONTENT_LENGTH*. Vor trebui citite atâtea caractere câte indică variabila de mediu *CONTENT_LENGTH*, nu mai multe.

Exemplul 10.2.5. Formularul următor permite introducerea unui titlu de proiect care va fi adăugat la sfârșitul unui fișier aflat pe server:

```
<form action="http://www.infoiasi.ro/cgi-bin/addproj.cgi"
  method="POST">
  <p>Introduceți titlul proiectului:</p>
  <input name="data" size="50" />
  <input type="submit" value="Trimite" />
</form>
```

Programul C care acceptă datele și le adaugă la fișierul *proj.txt* este următorul:

```
#include <stdio.h>
#include <stdlib.h>

#define MAXLEN 80
#define EXTRA 5
/* 4 pentru numele campului "data", 1 pentru "=" */
#define MAXINPUT MAXLEN+EXTRA+2
/* 1 pentru sfarsit de linie, 1 pentru NULL */

/* numele fisierului (cale relativa) */
#define DATAFILE "proj.txt"

/* functie de decodificare a datelor pasate programului CGI */
```

```
void unencode(char *src, char *last, char *dest)
{
    for (; src != last; src++, dest++)
        if (*src == '+') /* inlocuieste "+" cu spatiu */
            *dest = ' ';
        else
            if (*src == '%')
                ( /* inlocuieste hexa cu caracterul */
                 int code;
                 if (sscanf(src + 1, "%2x", &code) != 1)
                     code = '?';
                 *dest = code;
                 src += 2;
                )
            else /* copiaza celelalte caractere... */
                *dest = *src;
        *dest = '\n';
        *++dest = '\0';
}

/* functia principala */
int main(void)
{
    char *lenstr;
    char input[MAXINPUT], data[MAXINPUT];
    long len;

    /* scrie Content-type, urmat de setul de caractere (optional),
       in acest caz ISO-8859-2 (folosim diacritice) */
    printf("%s%c%c\n",
           "Content-type: text/html;charset=iso-8859-2", 13, 10);

    /* scrie antetul documentului HTML generat */
    printf("<html>\n<head><title>R&#259;spuns</title></head>\n");

    /* ia numarul de caractere transmise */
    lenstr = getenv("CONTENT_LENGTH");

    /* numarul de caractere este corect? */
    if (lenstr == NULL ||
        sscanf(lenstr, "%ld", &len) != 1 ||
        len > MAXLEN)
        printf("<p>Eroare - formular incorect?\n");
    else
    {
```

```

FILE *f;
/* citește de la intrarea standard datele */
fgets(input, len + 1, stdin);
/* decodifica linia */
unencode(input + EXTRA, input + len, data);
/* adaugă în fișier */
f = fopen(DATAFILE, "a");
if (f == NULL)
    printf("<p>Eroare la adăugarea &icirc;n" +
        "fi&#351; ier...");
else
    fputs(data, f);
fclose(f);
printf("<p>Mul&#355;umim! Linia <b>%s</b> a fost adăugată;
    ugat&#259; &icirc;n fi&#351;ier</p>", data);
}
/* sfârșitul documentului */
printf("</body>\n</html>\n");
return 0;
}

```

Propunem cititorului să conceapă, în limbajul C, scriptul CGI care să realizeze vizualizarea informațiilor despre proiecte, prin consultarea conținutului fișierului `proj.txt`.

10.3. Servleturi

Un **servlet** este o componentă software pe partea de server, scrisă în Java și care extinde dinamic funcționalitatea unui server (de obicei, un server HTTP). Reamintim că un applet este o componentă software pe partea clientului care rulează în cadrul unui navigator Web. Spre deosebire de appleturi, servleturile nu afișează o interfață grafică către utilizator, ci returnează doar rezultate către client (de obicei sub forma unui document HTML).

În fapt, servleturile sunt clase Java care se conformează unei interfețe specifice ce poate fi apelată de către server. Funcționalitatea furnizată de un servlet nu privește numai serverele Web, ci se poate extinde la orice server care suportă Java și Servlet API (cum ar fi: FTP, Telnet, mail, servere de știri). Servlet API este o specificație dezvoltată de Sun Microsystems (și alții) care definește clasele și interfețele necesare pentru crearea și execuția servleturilor.

Servleturile furnizează un cadru pentru crearea de aplicații care implementează paradigma cerere/răspuns. De exemplu, când un navigator trimite o cerere către server, serverul o trimite mai departe unui servlet. Servletul procesează cererea (eventual accesând o bază de date) și construiește un răspuns convenabil (de obicei în HTML) care este returnat clientului. Servleturile funcționează asemănător CGI-urilor.

10.3.1. Servlet API

Servlet API (*Application Programming Interface*) este o extensie Java standard a platformei Java 2, ediția standard (J2SE – *Java 2 Standard Edition*). Toate extensiile Java sunt conținute în pachetul `javax`. Pachetele pentru servleturi sunt `javax.servlet` și `javax.servlet.http`. Servlet API este disponibil gratuit ca:

- o bibliotecă de sine stătătoare Java (<http://java.sun.com/products/servlet/>);
- parte componentă a proiectului Apache Tomcat (<http://jakarta.apache.org/tomcat/>).

Pe de altă parte, spre deosebire de ediția standard, Servlet API este un membru obligatoriu al platformei Java 2, Enterprise Edition (J2EE) și este inclus în Java 2 SDK Enterprise Edition (<http://java.sun.com/j2ee/>).

10.3.2. Funcționalitatea servleturilor

Ca și CGI-urile, servleturile permit interacțiunea în ambele sensuri între client și server. Iată o parte dintre posibilitățile furnizate de servleturi:

- construiesc dinamic și returnează un document HTML pe baza cererii clientului;
- procesează datele completate de utilizatori printr-un formular HTML și returnează un răspuns;
- furnizează suport pentru autentificarea utilizatorului și alte mecanisme de securitate;
- interacționează cu resursele serverului, cum ar fi baze de date, fișiere cu informații utile pentru clienți;
- procesează intrările de la mai mulți clienți pentru aplicații, cum ar fi jocuri în rețea;
- permite serverului să comunice cu un applet client printr-un protocol specific și păstrează conexiunea în timpul conversației;
- atașează automat elemente de design pentru pagini Web, cum ar fi antete sau note de subsol, pentru toate paginile returnate de server;
- redirecționează cereri de la un server la altul în scop de echilibrare a încărcării (eng. *load-balancing*);
- partiționează un serviciu logic între servere sau între servleturi pentru a procesa eficient o problemă.

Există și alte soluții software care extind funcționalitatea unui server HTTP:

1. FastCGI, dezvoltat inițial de firma *Open Market*, îmbunătățește CGI-ul standard, însă nu se poate compara cu performanța și eficiența unui servlet Java. De exemplu, spre deosebire de servleturi, FastCGI necesită cel puțin un proces pentru fiecare program și fiecare proces poate servi doar o cerere la un moment dat.
2. NSAPI/ISAPI (*Netscape Server API/Microsoft Internet Server API*) sunt interfețe pentru extinderea funcționalității unui server HTTP. Acestea furnizează

mai multă performanță decât servleturile, însă sunt legate de servere HTTP și platforme particulare. De exemplu, ISAPI este suportat de *Microsoft Internet Information Server* (IIS) care rulează pe platforma *Windows NT/2000/XP/.NET*. Similar, aplicațiile NSAPI sunt compatibile doar cu câteva servere Web din *Netscape/iPlanet*. Chiar dacă serverele NSAPI sunt disponibile sub *Windows* și *UNIX*, din moment ce programele sunt scrise în C, aceste aplicații trebuie recompile pentru fiecare nouă platformă.

3. Alte soluții sunt oferite de PHP, Zope sau Lotus Notes/Domino.

10.3.3. Execuția servleturilor

Există multe posibilități de a executa servleturi. Anumite servere Web, cum ar fi *iPlanet Web Server* și *W3C Jigsaw*, oferă suport nativ pentru servleturi. Alte servere HTTP, cum ar fi *Apache Web Server* și *Microsoft Internet Information Server*, oferă suport prin produsele care se pot instala:

- *Tomcat* de la *Apache*;
- *JRun* de la *Allaire*;
- *ServletExec* de la *New Atlanta Communications*;
- *Resin* de la *Caucho Technology*.

În continuare, vom prezenta cel mai simplu server, care este și gratuit: *Apache Tomcat*. Acesta este o parte a proiectului *Apache Jakarta* și reprezintă o implementare de referință oficială pentru servleturile Java și pentru specificațiile JSP (eng. *Java Server Pages*). Poate fi obținut de la adresa:

■ <http://jakarta.apache.org/tomcat/>

Apoi, se instalează local dezarhivând fișierele copiate. Urmează pornirea serverului *Tomcat* prin comanda *startup*. Se poate verifica dacă instalarea și pornirea serverului *Tomcat* s-au făcut cu succes dacă atunci când se solicită unui navigator Web adresa (URL-ul) <http://localhost:8080/>, se obține pagina de lucru a serverului *Tomcat*. Se poate modifica numărul portului (8080) editând și schimbând porțiunea de cod corespunzătoare din fișierul */jakarta-tomcat/conf/server.xml*.

Lucrurile se petrec similar pe platforma *Unix/Linux*.

Toate servleturile specifice standardului *Servlet API 2.2* utilizează conceptul de aplicație Web (mai nou a apărut versiunea 3.0). O aplicație Web este o ierarhie de directoare și fișiere care formează împreună o aplicație. De obicei, acestea sunt distribuite într-un fișier de aplicații Web arhivate (în general, cu extensia *.war*). Toate aplicațiile Web utilizează aceeași structură de directoare referitoare la serverul utilizat. Însă locul unde sunt instalate aplicațiile Web poate să difere de la un server la altul. De exemplu, *Tomcat* păstrează toate aplicațiile Web în directorul */jakarta-tomcat/webapps*, iar *JRun* păstrează aplicațiile Web pentru serverul implicit în */jrun/servers/default*.

În aceste subdirectoare ale directorului *webapps* va căuta serverul *Tomcat* fișiere HTML, JSP și imaginile asociate unei aplicații Web. În general, fiecărei aplicații Web i se atribuie un „drum de context” unic de către administratorul de sistem.

Toate cererile către acest drum de context vor fi direcționate către aplicația Web corespunzătoare.

Dacă aplicației Web i se atribuie drumul de context */examples*, atunci URL-ul <http://localhost:8080/examples/index.html> va afișa fișierul *index.html* aflat în directorul *jakarta-tomcat/webapps/examples*.

Drumul de context pentru fiecare aplicație Web este specificat în fișierul *server.xml* din directorul *jakarta-tomcat/conf*.

În plus, o aplicație Web poate fi definită prin specificarea unui drum de context vid. De exemplu, *Tomcat* instalează aplicația *ROOT* ca aplicație implicită prin asignarea contextului vid. De exemplu, URL-ul <http://localhost:8080/index.html> va returna fișierul *index.html* din directorul */jakarta-tomcat/webapps/ROOT*.

Apelarea servleturilor utilizând serverul *Tomcat* se realizează după următorul format pentru URL:

■ [http://<server>:<port>/<drum_context>/servlet/<nume_servlet>\[/<informatii_drum>\]\[?<sir_interogare>\]](http://<server>:<port>/<drum_context>/servlet/<nume_servlet>[/<informatii_drum>][?<sir_interogare>])

Identificatorii scriși între paranteze unghiulare (<, >) semnifică faptul că aceștia trebuie înlocuiți. *<drum_context>* reprezintă locul unde se memorează aplicația Web (aceasta trebuie să fie unic). Cuvântul *servlet* indică serverului *Tomcat* că este vorba de un *servlet* și nu o pagină HTML sau JSP; *<nume_servlet>* reprezintă numele clasei *servletului* sau numele *servletului* (alias); *<informatii_drum>* și *<sir_interogare>* sunt componente suplimentare pentru un URL. *<informatii_drum>* se referă la informații suplimentare. Acestea pot fi obținute aplicând metoda *getPathInfo()* a obiectului *HttpServletRequest*. De exemplu, dacă se apelează un *servlet* numit *ObțineCaleServlet* prin URL-ul:

■ <http://localhost:8080/servlet/ObțineCaleServlet/html/public?name=value>

atunci valoarea returnată de metoda *getPathInfo()* este */html/public*.

<sir_interogare> permite trimiterea de informații suplimentare unui *servlet* utilizând formatul *nume=valoare*. Șirurile *nume=valoare* din *interogare* (eng. *query*) pot fi citite apelând pentru un obiect *HttpServletRequest* metodele *getParameterNames()*, *getParameter()* și/sau *getParameterValue()*.

10.3.4. Structura de bază a unui servlet

Cel mai ușor mod de definire a servleturilor este prin extinderea uneia dintre cele două clase de bază referitoare la servleturi: *GenericServlet* și *HttpServlet*. De fapt, nu este obligatorie moștenirea acestor clase, ci este suficientă implementarea interfeței *Servlet*.

Toate servleturile suprascriu cel puțin o metodă necesară funcționalității lor. Metoda care este automat apelată ca răspuns la cererea fiecărui client se numește *service()*. Această metodă poate fi suprascrisă pentru a furniza o funcționalitate

implicită. Cu toate acestea, servleturile care extind `HttpServlet` pot să nu suprascrie metoda `service()`. În acest caz, implementarea implicită a acestei metode va apela automat altă metodă pentru a răspunde cererii clientului.

Mai există încă două metode implementate de majoritatea servleturilor: `init()` și `destroy()`. Metoda `init()` se apelează o singură dată, când este încărcat servletul (similar cu un constructor). Metoda `destroy()` este apelată când servletul este descărcat, eliberând resursele servletului.

Un schelet pentru servleturi poate fi (fără a preciza parametrii și excepțiile):

```
public class ScheletServlet extends HttpServlet {
    public void init() {
        // codul de initializare
    }
    public void service() {
        // partea efectiva de lucru a servletului
    }
    public void destroy() {
        // eliberarea resurselor
    }
} // sfarsitul clasei ScheletServlet
```

Dacă servletul extinde `HttpServlet`, dezvoltatorul servletului poate să nu suprascrie metoda `service()`, ci să implementeze altă metodă care va fi automat apelată de metoda moștenită `service()`. Metoda automat apelată de `service()` depinde de tipul cererii HTTP (de exemplu, `doGet()` este apelată pentru cererile GET și `doPost()` este apelată pentru cererile POST).

10.3.5. Ciclul de viață al unui servlet

În principiu, procesul apelării de către server a unui servlet se poate împărți în opt pași:

1. Serverul încarcă servletul când acesta este cerut de client sau la startarea serverului, dacă așa impune configurația. Servletul poate fi încărcat local sau dintr-o locație de la distanță folosind o facilități de încărcare a claselor Java. De exemplu, putem avea:

```
Class c = Class.forName("surseFlux.ServletUnu");
```

2. Serverul creează o instanță a clasei servletului pentru deservirea tuturor cererilor. Folosind fire de execuție multiple, cererile concurente pot fi deservite de o singură instanță a servletului. De exemplu, putem avea:

```
Servlet s = (Servlet) c.newInstance();
```

3. Serverul apelează metoda `init()` a servletului care garantează că termină execuția înaintea procesării primei cereri a servletului. Dacă serverul creează mai multe instanțe ale servletului, metoda `init()` se apelează de fiecare dată pentru fiecare instanță.

4. În momentul primirii unei cereri pentru servlet, serverul construiește un obiect `ServletRequest` sau `HttpServletRequest` din datele incluse în cererea clientului. De asemenea, acesta construiește un obiect `ServletResponse` sau `HttpServletResponse` care furnizează metode pentru returnarea răspunsului. Tipul parametrului depinde dacă servletul extinde `GenericServlet` sau respectiv `HttpServlet`.
5. Serverul apelează metoda `service()` (care pentru servleturi HTTP poate apela o metodă specifică, cum ar fi `doGet()` sau `doPost()`), trimițând ca parametri obiectele construite la pasul 4. Când sosesc cereri concurente, metodele `service()` pot rula simultan în fire de execuție separate (cu excepția situației când servletul implementează interfața `SingleThreadModel`).
6. Metoda `service()` procesează cererea clientului prin evaluarea obiectului `ServletRequest` sau `HttpServletRequest`. Apoi acesta răspunde utilizând obiectul `ServletResponse` sau `HttpServletResponse`.
7. Dacă serverul primește încă o cerere pentru acest servlet, procesul începe din nou la pasul 5.
8. De fiecare dată când containerul servletului determină că un servlet trebuie descărcat (din motive de corecție sau dacă acesta a „căzut”), serverul apelează metoda `destroy()` după terminarea firelor de execuție ale metodei `service()` sau după terminarea limitei de timp definite de server. Servletul poate starta dealocatorul de memorie (*garbage collection*).

Vom prezenta mai întâi un servlet generic. Spre deosebire de servleturile HTTP, servleturile generice nu sunt specifice unui protocol anume. Servleturile generice (care extind clasa `GenericServlet`) sunt utilizate de obicei în aplicații de rețea care implică un protocol obișnuit sau orice alt protocol diferit de HTTP, cum ar fi FTP, SMTP sau POP3. Exemplul de mai jos conține un servlet generic care întoarce timpul curent de la server.

Exemplul 10.3.1. Servlet generic.

```
import javax.servlet.*;

/** Acest servlet returneaza timpul curent de la server */
public class ServletDeTimp extends GenericServlet {

    /** Serverul apeleaza metoda service() pentru a raspunde
     la cererile adresate servletului */
    public void service(ServletRequest cerere, ServletResponse
        raspuns)
        throws ServletException, java.io.IOException {
        // declarăm o referință către fluxul de ieșire al clientului
        java.io.PrintWriter iesireClient = raspuns.getWriter();
        // tipărim timpul curent și data către fluxul de ieșire
        iesireClient.println(new java.util.Date());
    }
}
```

```
iesireClient.close();
```

Fișierul `ServletDeTemp.class` va fi stocat în directorul `jakarta-tomcat/webapps/examples/Web-inf/classes`, iar în navigatorul Web se va insera adresa `http://localhost:8080/examples/servlet/ServletDeTemp`. Servletul va afișa în fereastra navigatorului Web timpul în care a fost executat servletul.

10.3.6. Clasa `HttpServlet`

Servleturile HTTP (cele care extind clasa `HttpServlet`) sunt utilizate pentru construirea de aplicații Web care de obicei întorc documente Web (HTML, XHTML, WML, XML etc.) ca răspuns la cererile navigatorului.

Clasa `HttpServlet` extinde clasa `GenericServlet`, deci moștenește toate funcționalitățile acesteia. În plus, `HttpServlet` adaugă funcționalitatea specifică protocolului HTTP și furnizează un cadru în care putem construi aplicații HTTP.

`HttpServlet` este o clasă abstractă prezentă în pachetul `javax.servlet.http`. Pentru că este abstractă, ea nu poate fi instanțiată. Când se construiește un servlet HTTP, trebuie extinsă clasa `HttpServlet`. Un servlet HTTP funcțional trebuie să implementeze una din metodele: `service()`, `doGet()`, `doHead()`, `doPost()`, `doPut()`, `doDelete()` corespunzătoare metodelor protocolului HTTP.

Metodele `init()`, `destroy()` și `getServletInfo()` aparțin clasei `GenericServlet`, din care se derivează clasa `HttpServlet`.

Metoda `service()` are prototipul:

```
public void service(HttpServletRequest request,
                    HttpServletResponse response) throws ServletException,
                    java.io.IOException;
```

Primul argument al metodei `service()` este un obiect care implementează interfața `HttpServletRequest` și este reprezentarea cererii clientului. Toate datele relevante din cerere (cum ar fi antetele HTTP) sunt încapsulate într-un obiect `HttpServletRequest`. Al doilea parametru este un obiect care implementează interfața `HttpServletResponse` și este reprezentarea răspunsului serverului către client.

Când se utilizează servleturi HTTP, este recomandată utilizarea metodelor de cerere specifice HTTP, cum ar fi `doGet()` sau `doPost()`, în locul suprascrierii metodei `service()`.

În general, metoda `service()` nu se suprascrie. Asta, deoarece clasa `HttpServlet` definește metode care se apelează pentru a răspunde diferitelor tipuri specifice de cereri HTTP. Dacă metoda `service()` nu este suprascrisă, atunci implementarea implicită este să se apeleze o metodă `doxxx()` care corespunde tipului cererii realizate. Dacă metoda `service()` este suprascrisă și sunt implementate anumite metode `doxxx()`, este sarcina dezvoltatorului de servlet să evalueze cererea HTTP și să apeleze metoda `doxxx()` corespunzătoare. Un container de servleturi nu apelează direct metodele `doxxx()`, ci metoda `service()`, care poate la rândul ei să apeleze o metodă `doxxx()`.

Dacă se încearcă apelarea unei metode `doxxx()`, care însă nu este suprascrisă, atunci se primește la client un mesaj de eroare: HTTP 400 "Bad Request".

Exemplul 10.3.2. Supradefinirea metodei `service()`:

```
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class Service extends HttpServlet
{
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        // setam tipul MIME pentru antetul HTTP
        response.setContentType("text/html");
        // obținem o referință la fluxul de ieșire
        PrintWriter iesire = response.getWriter();
        // determinăm tipul cererii
        if (request.getMethod().equalsIgnoreCase("GET")) {
            iesire.println(
                "O cerere GET rezolvată prin metoda service()");
        }
        else { // nu este o cerere GET
            // trimitem un mesaj de eroare din moment ce doar cererile
            // GET sunt suportate
            response.sendError(HttpServletResponse.SC_BAD_REQUEST);
        }
        iesire.close(); // închidem fluxul de ieșire
    }
}
```

Metoda `doGet()` are prototipul:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException,
                     java.io.IOException;
```

Metoda `doGet()` este apelată de implementarea implicită a metodei `service()` ca răspuns la o cerere HTTP de tip GET. Ca și alte metode de tip `doxxx()`, metoda `doGet()` are ca parametri o reprezentare a cererii clientului și un răspuns al serverului. Citirea obiectului cerere și formarea (trimiterea) obiectului răspuns este funcția principală a unui servlet.

Exemplul 10.3.3. Procesarea unei cereri prin metoda GET:

```
import javax.servlet.http.*;
import javax.servlet.ServletException;
```

```
import java.io.*;

public class GetUnu extends HttpServlet
{
    /** Returnam un mesaj scurt catre client */
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        // setam tipul MIME pentru antetul HTTP
        response.setContentType("text/html");
        // obtinem o referinta la fluxul de iesire
        PrintWriter iesire = response.getWriter();
        // determinam tipul cererii
        iesire.println(
            "O cerere GET rezolvata prin metoda doGet()");
        iesire.close(); // inchidem fluxul de iesire
    }
}
```

Metoda `doHead()` a fost eliminată din Servlet API 2.2 și reintrodusă în Servlet API 2.3 pentru a rezolva cererile HTTP de tip HEAD. Dacă metoda `doHead()` nu este suprascrisă de servlet, implementarea implicită va apela metoda `doGet()` pentru a genera câmpuri antet corespunzătoare. Apoi, se va returna către client un răspuns ce conține doar antetul HTTP (nu și conținutul documentului generat).

Continuăm cu prezentarea metodei `doPost()` care are prototipul:

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    java.io.IOException;
```

Metoda `doPost()` este apelată de fiecare dată când apare o cerere HTTP de tip POST primită de servlet. Această metodă este de obicei utilizată pentru procesarea informațiilor colectate dintr-un formular Web. Informația introdusă de utilizator într-un formular HTML este încapsulată într-un obiect `HttpServletRequest` și trimisă metodei `doPost()`. Dacă nu este găsită o implementare a metodei `doPost()`, atunci se trimite către client un mesaj de eroare: HTTP 400 "Bad Request".

Pentru a remarca funcționalitatea unui servlet, vom prezenta două exemple de utilizare a metodelor `doGet()` și `doPost()`.

Exemplul 10.3.4. Presupunem că am scris în fișierul `formular.html` următorul cod XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
<title>Un exemplu de adunare</title>
<!-- Specificarea setului de caractere pentru diacritice -->
<meta http-equiv="Content-Type"
    content="text/html; charset=iso-8859-2" />
</head>

<body>
    <h1>Suma a dou&#259; numere reale</h1>
    <p>Da&#355;i cele doua numere! :)</p>
    <form method="GET"
        action="http://localhost:8080/examples/servlet/SumaGet">
        <p>Num&#259;rul 1: <input type="text" name="nr1" /></p>
        <p>Num&#259;rul 2: <input type="text" name="nr2" /></p>
        <p><input type="submit" value="Trimite" />
            &nbsp;  <input type="reset" value="&#350;terge" /> </p>
    </form>
</body>
</html>
```

Se observă că acest fișier utilizează metoda GET. De exemplu, sub Windows, utilizând serverul *Web Tomcat*, plasăm în directorul `C:\Tomcat\jakarta-tomcat\webapps\examples\Web-inf\classes\` fișierul `.class` corespunzător codului sursă Java:

```
import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class SumaGet extends HttpServlet {
    // procesarea formularului XHTML
    protected void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        String numarUnu = request.getParameter("nr1");
        String numarDoi = request.getParameter("nr2");
        double nr1 = 0.0, nr2 = 0.0;
        Double d1Temp = Double.valueOf(numarUnu);
        nr1 = d1Temp.doubleValue();
        d1Temp = Double.valueOf(numarDoi);
        nr2 = d1Temp.doubleValue();
        double suma = nr1 + nr2;

        PrintWriter iesire = response.getWriter();
        iesire.println("<html><head>");
        iesire.println("<title>Un exemplu de adunare</title>");
        iesire.println("</head>");
```



```

    iesire.println("<body>");
    iesire.println("<h1>Suma celor doua numele</h1>");
    iesire.println("<p>Suma celor doua numere reale este " +
        suma + "</p>");
    iesire.println("</body></html>");
    iesire.close(); // inchidem fluxul de iesire
}
}

```

Astfel, deschizând documentul .html de mai sus într-un navigator Web, după completarea câmpurilor și apăsarea butonului *Submit*, se va apela servletul de mai sus apelând metoda `doGet()`, care va trimite către client o pagină Web care va afișa suma celor două numere introduse în formular.

În continuare, vom prezenta o variantă mai elegantă de a realiza suma a două numere. Astfel, vom vedea o funcționalitate mai mare a servletului utilizând ambele metode: `doGet()` și `doPost()`.

Exemplul 10.3.5. Utilizarea celor două metode de prelucrare a formularelor: GET și POST.

```

import javax.servlet.http.*;
import javax.servlet.ServletException;
import java.io.*;

public class Suma extends HttpServlet {

    // pentru metoda GET
    // returnam un formular HTML pentru citirea a doua numere
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // setam tipul MIME pentru antetul HTTP
        response.setContentType("text/html");

        // scriem documentul Web cu formularul
        PrintWriter iesire = response.getWriter();
        iesire.println("<html><head>");
        iesire.println("<title>Un exemplu de adunare</title>");
        iesire.println("</head>");
        iesire.println("<body>");
        iesire.println("<h1>Suma a doua numere</h1>");
        iesire.println("<p>Dati cele doua numere double</p>");
        iesire.println("<form method=\"POST\" action=\"" +
            "\"http://localhost:8080/examples/servlet/Suma\">");

```

```

    iesire.println(
        "<p>Numarul 1: <input type=\"text\" name=\"nr1\"></p>");
    iesire.println("<p>Numarul 2: <input type=\"text\" " +
        "name=\"nr2\"></p>");
    iesire.println(
        "<p><input type=\"submit\" value=\"Trimite\">&nbsp; " +
        "<input type=\"reset\" value=\"Sterge\"></p>");
    iesire.println("</form></body></html>");
    iesire.close(); // inchidem fluxul de iesire
}

// pentru metoda POST
// procesam datele primite din formularul HTML si returnam
// un document HTML care afiseaza suma celor doua numere
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException {
    // setam tipul MIME pentru antetul HTTP
    response.setContentType("text/html");

    // preluam parametrii
    String numarUnu = request.getParameter("nr1");
    String numarDoi = request.getParameter("nr2");
    double nr1 = 0.0, nr2 = 0.0;
    Double d1Temp = Double.valueOf(numarUnu);
    nr1 = d1Temp.doubleValue();
    Double d2Temp = Double.valueOf(numarDoi);
    nr2 = d2Temp.doubleValue();

    // efectuam suma
    double suma = nr1 + nr2;

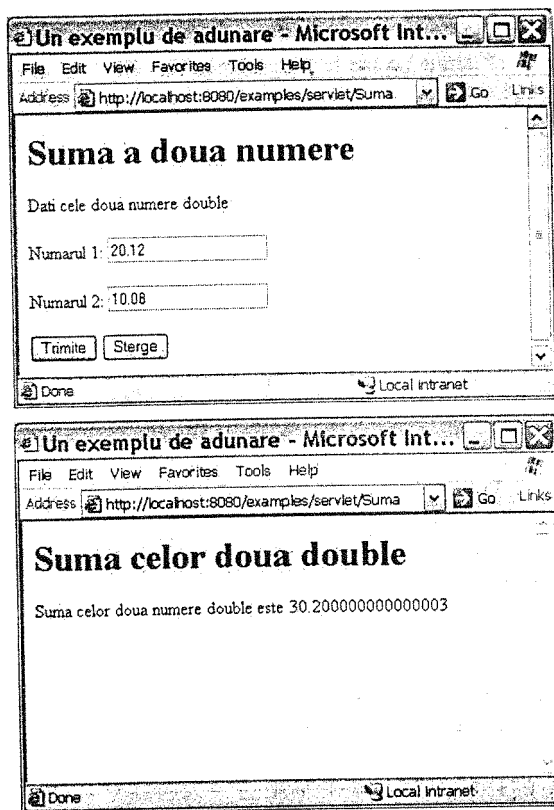
    // scriem rezultatul
    PrintWriter iesire = response.getWriter();
    iesire.println("<html><head>");
    iesire.println("<title>Un exemplu de adunare</title>");
    iesire.println("</head>");
    iesire.println("<body>");
    iesire.println("<h1>Suma celor doua double</h1>");
    iesire.println("<p>Suma celor doua numere double este " +
        suma + "</p>");
    iesire.println("</body></html>");
    iesire.close(); // inchidem fluxul de iesire
}
} // sfarsitul definitiei clasei Suma

```

În exemplul acesta, metoda `doGet()` este utilizată pentru crearea unui formular Web. La apăsarea butonului de submit, datele formularului sunt trimise către servlet și procesate de metoda `doPost()`. Aceasta construiește un document HTML care conține suma celor două numere trimise și apoi le returnează clientului.

De altfel, când se dorește citirea de informații de la client, se procedează de obicei în acest fel: utilizând metoda `doGet()`, se creează un formular HTML și apoi se prelucrează datele primite returnându-se o pagină Web cu ajutorul metodei `doPost()`.

Execuția servletului poate fi împărțită în două etape, trimiterea datelor și primirea rezultatului, astfel:



10.3.7. Redirectarea cererii

Redirectarea unei cereri constă în trimiterea unui URL ca răspuns, iar navigatorul Web va face apel la resursa respectivă pentru a afișa un rezultat. Acest lucru se poate realiza prin apelul metodei `sendRedirect()` din interfața `HttpServletResponse`. Această metodă are un parametru de tip `String` care conține URL-ul la care se realizează redirectarea.

Exemplul 10.3.6. Redirectarea cererii la adresa:

```
http://localhost:8080/examples/servlet/ServletDeTemp;
import javax.servlet.*;
import javax.servlet.http.*;

public class Redirectare extends HttpServlet {
    /** Tratarea cererilor prin metoda GET */
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException {
        /** redirectarea cererii */
        response.sendRedirect(
            "http://localhost:8080/examples/servlet/ServletDeTemp");
    }
}
```

Dacă în navigatorul Web se va scrie adresa `http://localhost:8080/examples/servlet/Redirectare`, atunci se va face apel la servletul `http://localhost:8080/examples/servlet/ServerDeTemp` (adresă care va apărea și în bara de adrese a navigatorului).

10.3.8. Returnarea unei erori

Servleturile pot întoarce coduri de eroare conform protocolului HTTP. Fiecare cerere HTTP primește ca răspuns un cod de stare care va indica tipul de răspuns primit (dacă este vorba de o informare, de succes sau eșec etc.). Pentru mai multe detalii despre codurile HTTP se poate consulta [PSW02], paginile 40-41 sau specificațiile protocolului HTTP.

Pentru a întoarce un astfel de cod de stare, interfața `HttpServletResponse` pune la dispoziție metoda `sendError()`. Aceasta are două forme:

- `void sendError(int cod)`
- `void sendError(int cod, String mesaj)`

unde `cod` este codul de stare, iar `mesaj` este mesajul care însoțește respectivul cod.

Exemplul 10.3.7. Trimiterea unui cod de stare.

```
import javax.servlet.*;
import javax.servlet.http.*;

public class Eroare extends HttpServlet {
    /** Procesarea cererilor prin metoda GET */
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException {
```

```

    /** trimiterea codului de stare 404 */
    raspuns.sendError(404);
}

```

Servletul de mai sus returnează codul de stare 404. Acesta corespunde situațiilor când URL-ul indicat este invalid (nu se află nici o resursă la adresa indicată).

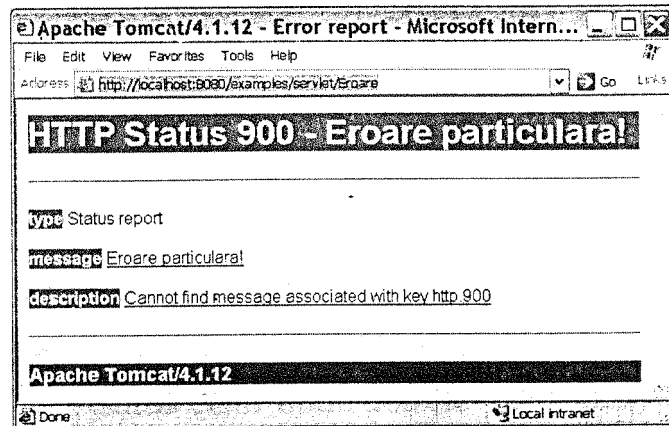
Se pot utiliza și coduri neutilizate în specificațiile protocolului HTTP. Spre exemplu, dacă înlocuim conținutul metodei `doGet()` cu următoarea linie:

```

    raspuns.sendError(900, "Eroare particulara!");

```

atunci se va obține un cod care nu va putea fi interpretat de navigatoarele Web și va fi afișat asemeni unui mesaj de eroare. Codul 900 va fi însoțit și de mesajul specificat, după cum se poate vedea și în figura de mai jos:



10.3.9. Exemplu de aplicație Web

În această secțiune dăm un exemplu de aplicație Web simplă implementată cu ajutorul servleturilor. Aplicația permite stocarea informațiilor despre persoane (nume, data nașterii, adresă și număr de telefon) prin intermediul unui formular Web. Un servlet va prelucra datele din respectivul formular și va înregistra datele într-un document XML. Pentru vizualizarea informațiilor existente am construit un servlet care returnează un document HTML cu datele referitoare la persoanele născute într-o anumită lună. Luna implicită este ianuarie, iar utilizatorul poate selecta luna dorită prin intermediul unui formular Web.

Codul XHTML pentru formularul de adăugare a informațiilor este următorul:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

```

```

<html>
<head>
    <title>Adaugare informatii</title>
    <!-- Specificam setul de caractere pentru diacritice -->
    <meta http-equiv="Content-Type"
        content="text/html; charset=iso-8859-2" />
</head>

<body>
<h2 align="center">Ad&#259;ugare informa&#355;ii</h2>
<form action="http://localhost:8080/examples/servlet/Adaugare"
    <del>method="POST"</del>
    <table border="0" align="center" cellpadding="3"
        cellspacing="10">
    <tr>
        <th>Numele &#351;i prenumele:</th>
        <td><input type="text" size="30" name="nume" /></td>
    </tr>
    <tr>
        <th>Data na&#351;terii (zi, lun&#259;, an):</th>
        <td>
            <input type="text" size="2" maxlength="2" name="zi" />&nbsp;
            <input type="text" size="2" maxlength="2" name="luna" />&nbsp;
            <input type="text" size="4" maxlength="4" name="an" />&nbsp;
        </td>
    </tr>
    <tr>
        <th>Adresa:</th>
        <td><textarea name="adresa" cols="30" rows="5"></textarea> </td>
    </tr>
    <tr>
        <th>Telefon:</th>
        <td><input type="text" size="30" name="telefon" /></td>
    </tr>
    <tr>
        <th colspan="2">
            <input type="submit" value="&Icirc;nregistrare" />
        </th>
    </tr>
    </table>
</form>
</body>
</html>

```

În navigatorul Web, acest formular va arăta astfel:

Acest document poate fi plasat în orice director de pe mașina curentă (locală). La apăsarea butonului de înregistrare se va apela servletul Adaugare, care are următorul conținut:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Date;
import java.util.Collection;
import java.util.Vector;
import java.util.Iterator;
import java.text.SimpleDateFormat;
import java.util.GregorianCalendar;
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;

public class Adaugare extends HttpServlet {
    /** Declarația datelor membre */
    private ServletContext context = null;

    /** metoda de initializare */
    public void init(ServletConfig config) throws
        ServletException {
        super.init(config);
        this.context = config.getServletContext();
    }
}
```

```
/** Procesarea informatiilor primite prin metoda POST */
protected synchronized void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
```

```
    /** Stabilirea tipului de document returnat */
    response.setContentType("text/html");
    /** Declararea unei variabile auxiliare */
    String mesaj = "Datele au fost inregistrate.";
    /** Preluarea parametrilor */
    String nume = request.getParameter("nume");
    String zi = request.getParameter("zi");
    String luna = request.getParameter("luna");
    String an = request.getParameter("an");
    String adresa = request.getParameter("adresa");
    String telefon = request.getParameter("telefon");
    /** Crearea unei colectii de persoane */
    Collection persons = new Vector();
    Persoana pers = null;
```

```
try {
    String path = context.getRealPath("/");
    try {
        /** mai intai se citesc datele din fisier... */
        FileInputStream fis = new FileInputStream(path +
            "temp/date.xml");
        XMLDecoder decoder = new XMLDecoder(
            new BufferedInputStream(fis));
        while (null != (pers = (Persoana) decoder.readObject())) {
            /** ... si se adauga in colectie */
            persons.add(pers);
        }
        decoder.close(); // inchiderea fisierului
    } catch (Exception ex) { }

    /** Validarea datei */
    int dd = Integer.parseInt(zi);
    if ( !((0 < dd) && (dd <= 31))) {
        throw new NumberFormatException("Ziua incorecta");
    }
    int mm = Integer.parseInt(luna);
    if ( !((0 < mm) && (mm <= 12))) {
        throw new NumberFormatException("Luna incorecta");
    }
}
```

```

int yyyy = Integer.parseInt(an);
if ( !(1900 < yyyy) ) {
    throw new NumberFormatException("An incorect");
}

/** Crearea unui obiect Persoana */
SimpleDateFormat dateFormatter = new SimpleDateFormat(
    "dd/MM/yyyy");
pers = new Persoana(numa,
    dateFormatter.parse(zi + "/" + luna + "/" + an),
    adresa, telefon);
/** adaugarea la colectie */
persons.add(pers);

/** suprascrierea fisierului */
FileOutputStream fos = new FileOutputStream(path +
    "temp/date.xml", false);
XMLEncoder encoder = new XMLEncoder( new BufferedOutput
Stream(fos));

/** salvarea datelor in fisier */
for(Iterator it = persons.iterator(); it.hasNext();) {
    encoder.writeObject((Persoana) it.next());
}
encoder.close();

/** Tratarea exceptiilor */
} catch (NumberFormatException e) {
    mesaj = "Data este incorecta!";
} catch (SecurityException e) {
    mesaj = "Nu este permisa scrierea pe server!";
} catch (Exception ex) {
    mesaj = ex.getMessage();
}

/** Returnarea raspunsului pentru navigator */
PrintWriter iesire = raspuns.getWriter();
iesire.println("<html><head>");
iesire.println("<title>Rezultat adaugare</title>");
iesire.println("</head>\n<body>");
iesire.println("<h3>" + mesaj + "</h3>");

/** Afisam datele primite */
iesire.println("<pre>");
iesire.println(numa);

```

```

iesire.println(zi);
iesire.println(luna);
iesire.println(an);
iesire.println(adresa);
iesire.println(telefon);
iesire.println("</pre>");

iesire.println("</body></html>");
iesire.close(); //inchiderea fluxului de iesire
}

/** oferirea informatiilor referitoare la servlet */
public String getServletInfo() {
    return "Adaugare data nasterii. vl.0";
}

/** metoda apelata la distrugerea servletului */
public void destroy() {
    context = null;
}
}

```

Pentru adăugarea unei persoane s-au citit mai întâi datele existente în documentul XML, s-a adăugat persoana în colecția de persoane, după care s-a salvat din nou întreaga colecție. Acest lucru ar putea constitui un dezavantaj în cazul în care ar fi un număr mare de persoane, adăugarea realizându-se lent. O rezolvare ar fi stocarea în documente separate a persoanelor născute într-o anumită lună (sau chiar într-o anumită zi din an, când numărul de persoane dintr-o lună este mare). Cititorul va modifica respectivul exemplu în funcție de numărul de persoane pe care dorește să le gestioneze.

Metoda de `doPost()` este declarată ca fiind `synchronized` pentru a nu permite actualizarea simultană a datelor de către două fire de execuție ale servletului (două cereri de adăugare). Altfel, ar putea apărea probleme la scrierea datelor în fișier sau pierderea de date. Dacă se încearcă vizualizarea persoanelor dintr-o anumită lună în timp ce lista acestora este actualizată, este posibil să apară o afișare necorespunzătoare sau o excepție de intrare/ieșire. Lăsăm în sarcina cititorului de a rezolva această problemă: de a nu permite citirea datelor în timp ce acestea sunt modificate.

Pentru a gestiona mai ușor datele despre persoane, am construit clasa `Persoana`:

```

import java.util.Date;
import java.io.Serializable;

public class Persoana implements Serializable {
    private String nume, adresa, telefon;
    private Date dataNasterii;

```

```

/** constructor fara parametri */
public Persoana() {
    nume = "";
    adresa = "";
    telefon = "";
    dataNasterii = null;
}

/** constructor cu parametri */
public Persoana(String nume, Date dataNasterii,
    String adresa, String telefon) {
    this.nume = nume;
    this.dataNasterii = dataNasterii;
    this.adresa = adresa;
    this.telefon = telefon;
}

/** metode pentru setarea valorilor */
public void setNume(String nume) { this.nume = nume; }
public void setDataNasterii(Date dataNasterii) {
    this.dataNasterii = dataNasterii;
}
public void setAdresa(String adresa) { this.adresa = adresa; }
public void setTelefon(String telefon) {
    this.telefon = telefon;
}

/** Metode pentru obtinerea informatiilor private */
public String getNume() { return nume; }
public Date getDataNasterii() { return dataNasterii; }
public String getAdresa() { return adresa; }
public String getTelefon() { return telefon; }

/** Generarea unei linii de tabel HTML */
public String linieTabel() {
    /** crearea unui calendar */
    java.util.Calendar cal = java.util.Calendar.getInstance();
    /** stabilim data nasterii pentru calendar */
    cal.setTime(dataNasterii);
    /** returnam codul HTML pentru o linie de tabel */
    return "<tr><td>" + nume + "</td><td>" +
        ((null != dataNasterii) ?
            cal.get(cal.DAY_OF_MONTH) + "-" + (cal.get(cal.MONTH) + 1)
            + "-" + cal.get(cal.YEAR) : "") + "</td><td>" + adresa +
        "</td><td>" + telefon + "</td> </tr>";
}

```

Pentru vizualizarea persoanelor născute într-o anumită lună, am construit un servlet DataNasterii. Acesta preia numărul lunii din parametrul luna. Dacă acesta nu este specificat, va fi aleasă luna ianuarie. După tabelul cu persoanele născute în luna stabilită am adăugat un meniu din care să alegem altă lună.

Codul corespunzător servletului DataNasterii este următorul:

```

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Calendar;
import java.beans.XMLDecoder;

public class DataNasterii extends HttpServlet {
    /** declararea datelor membre */
    protected String [] LUNA = { "ianuarie", "februarie", "martie",
        "aprilie", "mai", "iunie", "iulie", "august", "septembrie",
        "octombrie", "noiembrie", "decembrie" };
    private ServletContext context = null;

    /** metoda de initializare a servletului */
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        this.context = config.getServletContext();
    }

    /** metoda de procesare a cererii */
    protected void procesareCerere(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException {
        /** stabilirea tipului documentului returnat */
        response.setContentType("text/html");
        /** Declararea variabilelor auxiliare */
        String mesaj = "Ok";
        StringBuffer buf = new StringBuffer();
        Persoana pers = null;
        Date data;
        Calendar cal = Calendar.getInstance();
        int l = 0;

        try {

```

```

/** preluarea parametrului */
String luna = cerere.getParameter("luna");
if (luna != null && luna.compareTo("") != 0)
    l = Integer.parseInt(luna);
/** daca este luna invalida alegem valoarea zero */
if (l < 0 && l > 11) l=0;
/** Extragerea datelor din fisier */
try {
    /** pregatirea fluxului de date */
    String path = context.getRealPath("/");
    FileInputStream fos = new FileInputStream(path +
        "temp/date.xml");
    XMLDecoder decoder = new XMLDecoder( new Buffered
        InputStream(fos));
    /** extragerea persoanelor */
    while(null !=(pers =(Persoana)decoder.readObject())) {
        /** Obtinem data nasterii */
        data = pers.getDataNasterii();
        if (null != data) {
            cal.setTime(data);
            /** Daca persoana s-a nascut in luna care prezinta
                interes... */
            if (cal.get(cal.MONTH)==l) /** ... o adaugam in
                tabel */
                buf.append(pers.linieTabel());
        }
    }
    /** inchiderea fluxului */
    decoder.close();
} catch (ArrayIndexOutOfBoundsException aie) {
}

/** Daca nu am gasit persoane nascute in luna specificata */
if (buf.length()==0)
    mesaj = "Nu am gasit persoane nascute in luna " +
        LUNA[l] + "!";

/** tratarea exceptiilor */
} catch (NumberFormatException e) {
    mesaj = "Parametru incorect!";
} catch (Exception e) {
    e.printStackTrace();
    mesaj = "Eroare interna!";
}

```

```

/** scrierea raspunsului */
PrintWriter iesire = raspuns.getWriter();
iesire.println("<html>");
iesire.println("<head>");
iesire.println("<title>Data Nasterii</title>");
iesire.println("</head>");
iesire.println("<body>");

if (mesaj.compareTo("Ok")==0) { /** daca nu s-au inregistrat
    erori */
    /** stabilirea titlului */
    iesire.println(
        "<h2 align=\"center\">Persoanele nascute in luna "
        + LUNA[l] + "</h2>");
    /** inceputul tabelului */
    iesire.println("<table width=\"100%\" border=\"1\" " +
        "cellspacing= \"0\"\" + "cellpadding=\"3\">");
    /** prima linie din tabel */
    iesire.println(
        "<tr><th>Nume si prenume</th> " +
        "<th>Data nasterii </th> " +
        "<th>Adresa</th><th>Telefon</th></tr>");
    /** Continutul tabelului */
    iesire.println(buf);
    /** sfarsitul tabelului */
    iesire.println("</table>");
} else { /** s-a gasit o eroare si o afisam */
    iesire.println(
        "<h2 style=\"color: red; text-align: center;\">
        + mesaj + "</h2>");
}

/** formularul pentru selectarea lunii */
iesire.println("<form action=\"\" +
    "/examples/servlet/DataNasterii\" method=\"POST\">");
iesire.println("<select name=\"luna\">");
for (int i=0; i<12; i++)
    iesire.println("<option value=\"\" + i + \"\"> " + LUNA[i] +
        "</option>");
iesire.println("</select>");
iesire.println(
    "&nbsp;<input type=\"submit\" value=\"Vizualizare\">");
iesire.println("</form>");

/** sfarsitul documentului */

```

```

    iesire.println("</body>");
    iesire.println("</html>");
    /** inchiderea fluxului de iesire */
    iesire.close();
}

/** tratarea metodei GET */
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
    procesareCerere(request, response);
}

/** tratarea metodei POST */
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, java.io.IOException {
    procesareCerere(request, response);
}

/** Informatii despre servlet */
public String getServletInfo() {
    return "Data Nasterii. v1.0";
}

/** metoda apelata la distrugerea servletului */
public void destroy() {
    context = null;
}
}

```

Rezultatul returnat de servletul de mai sus va fi de forma:

Nume si prenume	Data nasterii	Adresa	Telefon
Tabusca Nicolae	20-1-1978	Str. Cerna Nr.15 Bl. Q12 Iasi	0232-165091
Corodeanu Viorel	12-1-1947	Str. Flopilor Nr. 7 Bacau	0733-131011
Agapescu Constantin	31-1-1962	Str. Grigore Ureche Nr. 7 Bl. G17 Sc. 1 Et. 6 Ap. 50 Galati	0555-123991

ianuarie Vizualizare

10.4. Concluzii

Continuăm cu prezentarea unor aspecte de care trebuie să ținem seama:

- Performanța.** Servleturile rulează de obicei în același spațiu de proces ca și serverul și sunt încărcate doar o singură dată. Astfel, servleturile sunt capabile să răspundă mai rapid și eficient la cererile clienților. În contrast, CGI-urile trebuie să creeze câte un nou proces pentru deservirea unei cereri. Cu toate acestea, CGI-urile scrise în limbaje compilate pot fi mai rapide.
- Compilarea în byte coduri Java** oferă execuții rapide față de programele scrise în limbaje script. Multe din erori sunt îndepărtate încă de la compilare, deci servleturile sunt mai stabile. Același lucru se observă și la CGI-urile scrise în Perl, deoarece acestea sunt compilate foarte repede, după care sunt executate.
- Rezistența la blocarea sistemului.** Mașina virtuală Java nu permite servleturilor accesul direct la locațiile de memorie, deci se elimină posibilitatea blocării sistemului ca rezultat al accesului la memoria invalidă (pointerii în C puteau conduce la acest lucru). JVM va propaga o excepție sau va rezolva eroarea fără blocarea sistemului.
- Portabilitatea platformei.** Caracteristica Java „scris o dată, rulează oriunde” permite servleturilor să fie distribuite ușor, fără a rescrie codul pentru fiecare platformă. Servleturile operează identic, fără modificări, când rulează pe UNIX, Windows sau alt sistem de operare. Același lucru este valabil și pentru limbajele de tip script cum ar fi Perl și bash.
- Portabilitatea serverelor.** Ne referim la proprietatea rulării servleturilor pe orice server Web. La adresa <http://java.sun.com/products/servlet/runners.html> există o listă completă cu serverele care furnizează suport nativ pentru servleturi în produsele lor, cum ar fi: Apache Tomcat, Inprise Application Server, Oracle Application Server și altele. Serverele JRun (de la firma Allaire) și ServletExec (de la firma New Atlanta Communications) adaugă suport pentru multe servere Web, printre care: Microsoft Internet Information Server, Netscape Enterprise Server și Apache Web Server.
- Durabilitate.** Servleturile rămân în memorie până când există o instrucțiune specifică de distrugere a lor. Astfel, servleturile necesită doar o instanțiere pentru a satisface mai multe cereri. De exemplu, se obișnuiește ca la încărcarea unui servlet să se creeze și mai multe conexiuni la baze de date.
- Încărcare dinamică.** Ca și appleturile, servleturile pot fi dinamic încărcate local sau prin rețea. Încărcarea dinamică asigură pentru servleturile nefolosite neconsumarea resurselor prețioase. Ele sunt încărcate doar când este nevoie.
- Extensibilitatea.** Noile instrumente de dezvoltare, noile biblioteci de clase Java și driverele de baze de date sunt constant disponibile și astfel pot fi utilizate de servleturi. Același lucru se poate spune și despre CGI-urile scrise în Perl.
- Concurența.** Spre deosebire de C/C++, mecanismele de concurență (*thread*) sunt moștenite în Java și sintaxa sincronizării *thread*-urilor este consistentă pe

toate platformele. Natura concurențială a servleturilor Java permite rezolvarea cererilor clienților în fire de execuție separate într-un singur proces.

10. **Orientarea obiect.** Servleturile furnizează o arhitectură simplă și elegantă pentru dezvoltarea aplicațiilor de rețea. Asta, deoarece Servlet API încapsulează toate informațiile esențiale și funcționalitatea în obiecte bine construite. De exemplu, Servlet API furnizează clase care lucrează cu obiecte abstracte, cum ar fi: cereri, răspunsuri, sesiuni și *cookie*-uri. Limbajul Perl posedă suport pentru programarea orientată obiect, însă nu este complet orientat obiect.
11. **Independența de protocol.** De obicei, servleturile sunt utilizate pentru extinderea funcționalității serverelor Web. Servleturile sunt complet independente de protocol, acestea suportând comenzi FTP, SMTP, POP3, sesiuni Telnet, grupuri de știri NNTP sau orice alt protocol (fie standard, fie creat de programatori). Același lucru este valabil și pentru alte limbaje (spre exemplu, limbajul Perl).
12. **Securitate.** Deoarece servleturile sunt scrise în Java, nu sunt posibile accesările nevalide la memorie sau violări de tip. În plus, există trei proprietăți care fac servleturile mai sigure decât CGI-urile. În primul rând, servleturile folosesc un manager de securitate pentru crearea unor reguli de securitate. Un manager de securitate poate restricționa rețeaua sau accesul la un fișier pentru un servlet de neîncredere. Pe de altă parte, un manager de securitate poate acorda drepturi depline pentru un servlet de încredere. În al treilea rând, un servlet are acces la toate informațiile conținute în fiecare cerere a clientului. Aceste informații conțin date de autentificare HTTP. Când sunt utilizate în conjuncție cu protocoalele de securitate cum ar fi SSL, servleturile pot verifica identitatea fiecărui client. Dezavantajul este că o dată obținute fișierele cod binar Java (.class), acestea se pot decompila, ceea ce este mult mai dificil la CGI-urile scrise în C/C++.

10.5. Test grilă

Întrebarea 10.5.1. Care dintre următoarele afirmații sunt adevărate referitor la clasa unui servlet?

- a) Nu se poate redefini metoda `service()`.
- b) Nu se mai pot defini noi metode sau date membre.
- c) Toate datele trebuie să fie publice.
- d) Obligatoriu trebuie definită una dintre metodele `doGet()` și `doPost()`.

Întrebarea 10.5.2. Dacă un servlet redefiniște doar metoda `doGet()`, atunci:

- a) Un apel prin metoda `POST` va întoarce un mesaj de eroare.
- b) Metoda `doGet()` trebuie neapărat să întoarcă un rezultat.
- c) În mod necesar trebuie implementată metoda `service()`.
- d) Un apel prin metoda `HEAD` va întoarce un mesaj de eroare.

Întrebarea 10.5.3. Dacă un servlet redefinește metoda `service()`, atunci:

- a) Trebuie neapărat să redefinească metodele `init()` și `destroy()`.

- b) În mod necesar va apela metodele de tipul `doXXX()`.
- c) Se obține eroare la compilare.
- d) Nu se vor mai putea prelua parametrii transmiși de formularele Web.

Întrebarea 10.5.4. Care dintre următoarele afirmații sunt false?

- a) Toate serverele Web posedă suport nativ pentru servleturi.
- b) Pentru ca pe un calculator să se poată încărca un document Web generat de un servlet, acesta trebuie să aibă instalată mașina virtuală Java.
- c) Servleturile pot fi independente de protocolul HTTP.
- d) Servleturile sunt utilizate doar pentru prelucrarea formularelor Web.

Întrebarea 10.5.5. Rularea unui servlet se poate invoca astfel:

- a) Cu ajutorul programului `java`.
- b) Direct dintr-un navigator Web.
- c) Dintr-o pagină Web, doar dintr-un formular.
- d) Cu ajutorul aplicației `servletviewer`.

10.6. Exerciții propuse spre implementare

Exercițiul 10.6.1. Minicalculator. Scrieți un servlet care pentru metoda `GET` generează un formular cu un câmp text, iar pentru metoda `POST` evaluează expresia aritmetică inserată în respectivul câmp și întoarce un document Web cu rezultatul obținut.

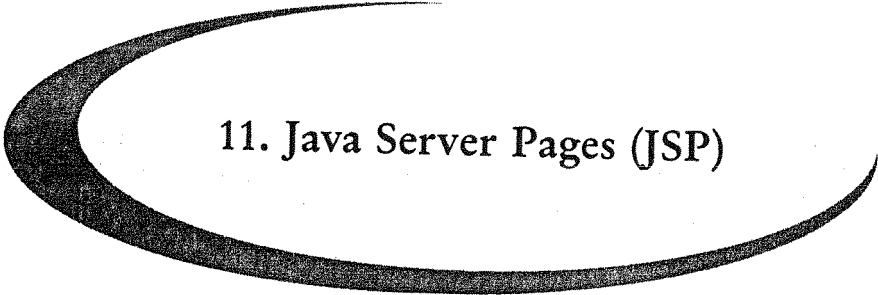
Exercițiul 10.6.2. Agendă telefonică. Să se ofere posibilitatea consultării unei agende telefonice (căutare după nume, număr telefon, adresă). Se va construi și un modul pentru administrarea agendei (eliminarea, adăugarea și respectiv modificarea înregistrărilor). Datele vor fi memorate într-un fișier XML.

10.7. Proiecte propuse spre implementare

Proiectul 10.7.1. Mersul trenurilor. Presupunem că dispunem de o bază de date (fie sub forma unui fișier, fie o bază de date SQL) care cuprinde mersul trenurilor din România. Scrieți un servlet care să modeleze completarea datelor despre toate rutele dintre două orașe (oraș plecare, oraș destinație, număr maxim de schimbări, ruta – opțional) din România și care să furnizeze o pagină Web ce listează toate posibilitățile existente cu informațiile aferente (timp, rută, schimbare, tip tren) sortate după numărul de ore și minute anticipate. Se va ține cont și de faptul că unele trenuri nu circulă sâmbăta/duminică. Se va implementa și un modul pentru actualizarea informațiilor referitoare la mersul trenului (adăugarea de noi trenuri, anularea celor existente, modificarea orei de plecare etc.).

Proiectul 10.7.2. Managementul situației școlare. Să se ofere posibilitatea elevilor/studenților de a consulta situația școlară pe Web. Se va da posibilitatea cadrelor

didactice de a nota studenții. Bineînțeles, fiecare profesor va trebui să pună note doar la disciplina sa și doar pentru clasele/grupele la care ține ore. Fiecare elev/student va putea vizualiza doar notele sale. Se va da posibilitatea generării de rapoarte pentru fiecare clasă, elev/student, disciplină/profesor sau pentru întreaga instituție (de exemplu, lista promovărilor/nepromovărilor).



11. Java Server Pages (JSP)

11.1. Cuvinte cheie

- JSP, servlet, pagini Web, server-side
- JavaBeans
- Tomcat

11.2. Introducere

Java Server Pages (JSP) este una dintre cele mai puternice tehnologii Web și este ușor de utilizat. JSP combină HTML și XML cu servleturile și tehnologia JavaBeans pentru a crea un mediu destul de productiv pentru dezvoltarea de situri Web independente de platformă și de o înaltă performanță.

Tehnologia JSP facilitează crearea conținutului dinamic pe partea de server a paginilor Web. Este asemănătoare cu ASP (Active Server Pages) de pe platforma Microsoft Windows și cu PHP (PHP: Hypertext Preprocessor), care este independent de platformă. JSP este o soluție Java pentru programarea pe partea de server, fiind o alternativă la CGI-urile clasice (Common Gateway Interface). JSP integrează numeroase tehnologii Java cum ar fi servleturile, JavaBeans și JDBC.

JSP extinde limbajul HTML oferind posibilitatea inserării de secvențe de cod Java prin intermediul unor taguri speciale. Programatorul are posibilitatea de a crea noi taguri și componente JavaBeans cu semnificațiile indicate de acesta. Astfel, se pot crea noi facilități pentru cei care se ocupă de partea de Web design.

În același timp, JSP este o extensie a servleturilor. În loc să scriem cod Java care să genereze pagini Web, vom crea pagini Web care vor conține elemente dinamice. Atunci când se primește o cerere de pagină JSP, se creează un servlet din respectiva pagină și acesta este executat, iar rezultatul este trimis ca răspuns la cererea primită.

Un avantaj important al JSP-urilor față de servleturi este faptul că se separă conținutul HTML static de cel dinamic. În cazul servleturilor, orice modificare minoră referitoare la designul paginii Web implică recompilarea respectivului servlet. La JSP-uri, partea de generare a conținutului dinamic este păstrată separat de cea statică prin utilizarea componentelor JavaBeans externe. Orice modificare a părții statice va fi vizibilă celor ce accesează respectivul JSP, întrucât la primirea cererii se recompilă automat și foarte repede pagina JSP, apoi se execută și rezultatul este trimis.

O dată scrisă, o pagină JSP poate fi stocată pe orice server Web (care are suport pentru JSP), oricare ar fi platforma pe care se află acesta, fără a suferi modificări.

Nu există limitări referitoare la tipul conținutului generat de părțile dinamice ale JSP-urilor. Acesta poate fi text obișnuit, HTML/DHTML, XML, WML, VRML etc.

JSP-urile sunt mai ușor de creat și pot avea funcționalitatea aproape a oricărui servlet. Servleturi sunt utilizate pentru a extinde funcționalitatea serverului Web (servicii de autentificare, validarea bazelor de date etc.) și pentru comunicarea cu appleturi sau alte aplicații Web.

11.3. Elemente JSP

11.3.1. Crearea paginilor JSP

Deoarece paginile JSP sunt executate pe partea de server, avem nevoie de un server Web. Pentru testarea exemplurilor vom utiliza serverul Tomcat. Paginile JSP vor fi stocate în directorul *tomcat_dir/webapps/Root*.

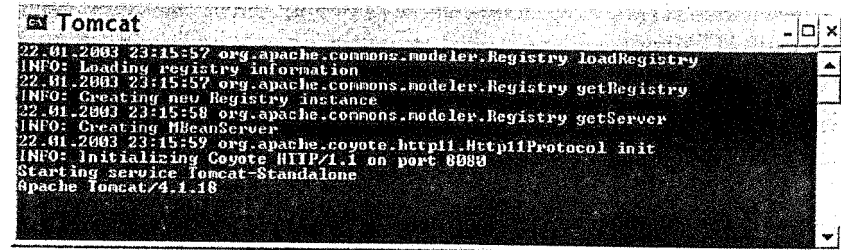
Paginile JSP au extensia *.jsp*. Orice pagină HTML poate fi pagină JSP.

Exemplul 11.3.1. Fie următoarea pagină simplă JSP, memorată în fișierul *test.jsp*:

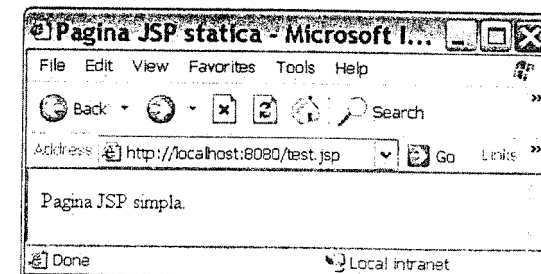
```
<html>
<head>
  <title>Pagina JSP statica</title>
</head>

<body>
  <p>Pagina JSP simpla.</p>
</body>
</html>
```

Acest fișier va fi stocat în directorul mai sus amintit. Se va porni serverul Tomcat și va apărea o fereastră care tot timpul va trebui să existe:



Apoi deschidem un navigator și scriem adresa corespunzătoare. Dacă rulăm local serverul pe o platformă Windows, vom introduce adresa *http://localhost:8080/test.jsp*. Rezultatul va fi:



Această pagină poate fi utilizată și ca test pentru a vedea dacă serverul suportă tehnologia JSP sau dacă este configurat corespunzător.

Exemplul 11.3.2. Următoarea pagină JSP afișează data curentă de pe serverul Web.

```
<%@ page import="java.util.Date" %>
<html>
<head>
  <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Data curenta de pe server: <b><%= new Date() %></b>.</p>
</body>
</html>
```

De asemenea, acest exemplu poate fi utilizat pentru testarea funcționalității corecte a serverului Web.

11.3.2. Comentarii

Există trei tipuri de comentarii care se pot utiliza în paginile JSP: comentarii HTML, comentarii JSP și comentarii Java.

Comentariile HTML încep cu simbolurile `<!--` și se termină cu `-->`. Exemplu de comentariu HTML:

```
<!-- Comentariu HTML -->
```

Aceste comentarii pot fi văzute atât de programator, cât și de utilizatori. Utilizatorii pot vedea comentariile HTML în momentul vizualizării sursei paginii HTML. Comentariile HTML pot conține taguri specifice JSP care vor fi înlocuite la momentul execuției de către server.

Exemplul 11.3.3. Fie următoarea pagină JSP:

```
<%@ page import="java.util.Date" %>
<!-- Data curenta de pe server: <%= new Date() %>. -->
<html>
<head>
  <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Vizualizati sursa acestei pagini (View source).</p>
</body>
</html>
```

La vizualizarea sursei HTML primite de la server va apărea:

```
<!-- Data curenta de pe server: Sat Oct 05 15:23:08 EEST
2002. -->
<html>
<head>
  <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Vizualizati sursa acestei pagini (View source).</p>
</body>
</html>
```

Observăm că serverul Web nu ignoră comentariile HTML.

Comentariile JSP apar între delimitatorii `<%--` și `-->`. Exemplu de comentariu JSP:

```
<%-- Comentariu JSP --%>
```

Comentariile JSP sunt eliminate în momentul transformării paginii JSP în servlet. Orice comentariu JSP va fi ignorat la transformarea paginii JSP în servlet și nu va fi vizibil utilizatorului. De aceea, se mai numesc comentarii ascunse.

Comentariile Java pot fi utilizate atunci când apar secvențe de cod Java. Acestea nu vor apărea în paginile Web trimise de către serverul Web, deoarece sunt eliminate la transformarea paginii JSP în servlet.

Exemplul 11.3.4. Utilizarea comentariilor Java:

```
<%@ page import="java.util.Date" %>
<%!
/* data este o variabila */
Date data = new Date();
%>
<html>
<head>
  <title>Pagina JSP dinamica</title>
</head>

<body>
  <p>Data curenta de pe server: <b><%= data %></b>.</p>
</body>
</html>
```

11.3.3. Directive

Directivele oferă posibilitatea de adăugare de informații adiționale și pentru descrierea atributelor paginii. De exemplu, directivele sunt utilizate pentru importarea pachetelor Java, includerea fișierelor și pentru accesarea librăriilor de taguri definite de utilizator.

Sintaxa generală pentru directive este:

```
<%@ directive [...] %>
```

sau utilizând spațiul de nume jsp:

```
<jsp:directive.directive [...] />
```

Pentru claritate se poate utiliza forma XML.

11.3.3.1. Directiva page

Directiva page poate fi utilizată pentru a importa pachete și clase Java, la fel ca instrucțiunea import (tot din Java). Exemplul 11.3.4. conține în prima linie o directivă page care importă clasa Date, specificându-se și pachetul din care face parte.

Directiva page poate poseda atributele din tabelul de mai jos:

Atribut	Valoare	Valoare implicită	Descriere
language	java	java	specifică limbajul de programare
extends	superclasa	depinde de platformă	indică superclasa pentru clasa servletului care se va genera
import	listă de pachete separate prin virgulă	java.lang.*, javax.servlet.http.*, javax.servlet.*, javax.servlet.jsp.*	importă lista de pachete și/sau clase specificate
session	true false	true	indică dacă se stabilește o sesiune
buffer	dimensiunea în kilo-octeți	8 sau mai mult	stabilește dimensiunea buffer-ului
autoFlush	true false	true	indică dacă se golește automat buffer-ul
isThreadSafe	true false	true	indică dacă poate fi accesată pagina de către mai multe fire de execuție simultan
info	text	șirul vid	specifică informații despre pagina JSP

Atribut	Valoare	Valoare implicită	Descriere
errorPage	un URL	nimic	indică URL-ul paginii care va fi trimisă utilizatorului în caz de eroare
isErrorPage	true false	false	indică dacă este pagina pentru erori
contentType	tipul MIME	text/html	specifică tipul documentului returnat

Observăm că directiva page stabilește informațiile privitoare la pagina JSP. De asemenea, observăm că valorile implicite sunt destul de rezonabile și rareori va trebui să le modificăm.

11.3.3.2. Directiva include

Directiva include conduce la includerea unui alt fișier în pagina JSP. Acest lucru este util atunci când dorim să creăm un sit unitar în care fiecare pagină să conțină același meniu de navigare. Respectivul meniu se va include într-un fișier separat (se va scrie secțiunea de cod HTML/JSP corespunzătoare meniului), iar acesta va fi inclus prin intermediul directivei include în cadrul tuturor paginilor. Dacă respectivul fișier conține elemente JSP, acestea vor fi și ele considerate în momentul transformării paginii JSP în servlet.

Directiva include are un singur atribut (file) care indică numele și locul unde se găsește respectivul fișier.

Exemplul 11.3.5. Pagina include.jsp va include fișierul meniu.txt. Ambele fișiere se află în același director. Conținutul paginii JSP este:

```
<html>
<head>
  <title>Test de incluziune</title>
  <meta http-equiv="Content-type"
    content="text/html; charset=ISO-8859-2" />
</head>

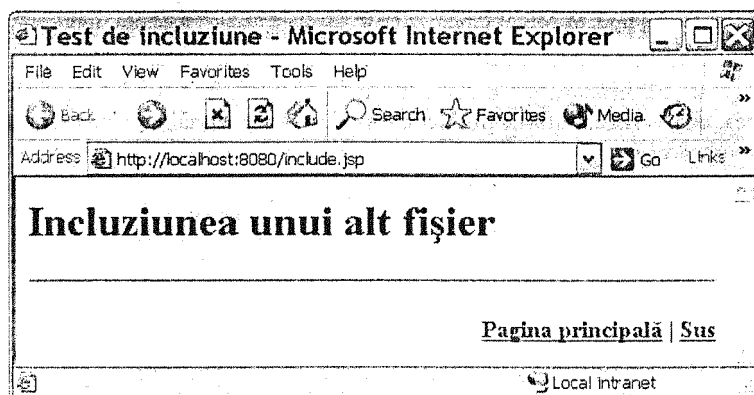
<body>
  <a name="sus"></a>
  <h1>Incluziunea unui alt fișier</h1>
  <%@ include file="meniu.txt" %>
</body>
</html>
```

iar al fișierului care va fi inclus:

```
<hr noshade="noshade" size="2" />
<h3 align="right">
```

```
<a href="index.html">Pagina principală</a> |
<a href="#sus">Sus</a>
</h3>
<% Comentariu JSP %>
```

În navigatorul Web, pagina JSP va apărea astfel:



Dacă vizualizăm pagina primită de navigator, vom observa dispariția comentariului JSP din fișierul inclus. Acest lucru denotă faptul că fișierele incluse sunt procesate și ele în momentul creării servletului din pagina JSP.

11.3.3.3. Directiva taglib

Crearea librărilor de taguri proprii este una dintre cele mai importante facilități pe care le oferă tehnologia JSP. Noi capacități ale paginilor JSP se pot încapsula prin intermediul tagurilor particularizate. Acestea pot fi utilizate de către persoanele care nu posedă cunoștințe de programare (cum ar fi cei care se ocupă de design). Astfel se separă partea de interfață de partea de funcționalitate.

Directiva taglib include o librărie de taguri definite de programatori. Aceasta conține două atribute: `uri`, care stabilește fișierul care conține definiția tagurilor, și `prefix`, care fixează un prefix unic pentru respectiva librărie de taguri, pentru a nu apărea conflicte la utilizarea unui același tag care este definit în două librării distincte. Prefixul se utilizează ca și spațiile de nume XML.

11.3.4. Declarații

În partea de declarații se pot declara date și funcții membre pentru utilizarea în interiorul paginilor JSP. Acestea vor face parte din servletul generat din pagina JSP. Declarațiile se pot insera astfel:

```
<%! declaratii %>
```

sau prin

```
<jsp:declaration> declaratii </jsp:declaration>
```

În exemplul 11.3.4. avem definită o variabilă de tip `Date` care ține data curentă.

Exemplul 11.3.6. În pagina JSP de mai jos s-a definit o dată membră privată și o metodă publică pentru obținerea valorii datei private.

```
<%@ page import="java.util.Date" %>
<%!
/* sectiune de declaratii */
private static String startTime = new Date().toString();
public static String getStartTime() { return startTime; }
%>
<html>
<head>
<title>Pagina JSP dinamica</title>
</head>

<body>
<p>Momentul la care JSP-ul a fost procesat de server:
<b><%=getStartTime() %></b></p>
</body>
</html>
```

11.3.5. Inițializarea și terminarea unui JSP

Înainte de execuție se va apela metoda `jspInit()` pentru realizarea de inițializări suplimentare, iar la terminarea execuției unui JSP se va apela metoda `jspDestroy()` pentru eliberarea unor resurse. Ambele metode nu posedă parametri, sunt publice și returnează `void`.

Exemplul 11.3.7. Pagina JSP va inițializa data membră privată `start` la inițializare și va atribui valoarea `null` la terminare.

```
<%!
private String start="";

public String comenteaza() {
return "<!-- Comentariu interior -->";
}

public void jspInit() {
start = "<!-- Comentariu la inceput -->";
}
```

```

}

public void jspDestroy() {
    start=null;
}

public String getStart() { return start; }
%>

<%= getStart() %>
<html>
<head>
<title>Initializare JSP</title>
</head>

<body>
<p>Continutul paginii.</p>
<%= comenteaza() %>
</body>
</html>

```

Navigatorul Web va primi următorul cod HTML:

```

<!-- Comentariu la inceput -->
<html>
<head>
<title>Initializare JSP</title>
</head>

<body>
<p>Continutul paginii.</p>
<!-- Comentariu interior -->
</body>
</html>

```

De aici se observă faptul că metoda `jspInit()` s-a apelat prima, întrucât variabila `start` a fost inițializată.

11.3.6. Obiecte implicite

Există numeroase obiecte implicite definite de arhitectura JSP. Acestea furnizează accesul la mediul din momentul execuției. Se poate întâmpla de multe ori să nu avem nevoie în mod direct de obiectele predefinite. Lista obiectelor predefinite, precum și o scurtă descriere a acestora se găsesc în tabelul următor:

Nume obiect	Clasa obiectului	Descriere
out	<code>javax.servlet.jsp.JspWriter</code>	Este utilizat în scriplet-uri sau trimis ca parametru altor metode.
request	<code>javax.servlet.ServletException</code>	Oferă toate informațiile privitoare la cererea primită sau la navigator.
response	<code>javax.servlet.ServletResponse</code>	Oferă acces la fluxul de ieșire a servletului.
session	<code>javax.servlet.http.HttpSession</code>	Este utilizat atunci când se dorește o pseudo-conexiune între client și serverul Web.
pageContext	<code>javax.servlet.jsp.PageContext</code>	Este util pentru accesarea mediului JSP și a componentelor JavaBeans.
config	<code>javax.servlet.ServletConfig</code>	Oferă informații despre proprietățile servletului.
page	<code>java.lang.Object</code>	Conține o referință la pagina JSP.
application	<code>javax.servlet.ServletContext</code>	Reprezintă aplicația Web utilizată pentru jurnalizarea (eng. <i>log</i>) paginii JSP.
exception	<code>java.lang.Throwable</code>	Este conținut doar de paginile de eroare și conține informații privind eroarea apărută.

11.3.7. Expresii

O expresie JSP este o expresie Java care este evaluată la momentul execuției, rezultatul fiind convertit la tipul `String` și scris în fluxul de ieșire. Sintaxa generală a expresiilor este:

```
<%= expresie_Java %>
```

respectiv

```
<jsp:expression> expresie_Java </jsp:expression>
```

Atenție! Expresia nu se va termina cu simbolul punct și virgulă.

Exemplele 11.3.2., 11.3.3., 11.3.4., 11.3.6. și 11.3.7. conțin expresii JSP.

11.3.8. Scriplet-uri

Până acum am văzut cum putem declara date și funcții membre, precum și inserarea expresiilor. Scriplet-urile permit adăugarea de cod Java îmbinat cu secvențe HTML. Sintaxa generală este următoarea:

```
<% cod_Java %>
```

respectiv

```
<jsp:scriptlet> cod_Java </jsp:scriptlet>
```

Exemplul 11.3.8. Pagina JSP va cerceta dacă navigatorul utilizat este Internet Explorer și va afișa un mesaj corespunzător.

```
<%
String navigator = (String) request.getHeader("user-agent");
%>
<html>
<head>
<title>Detectare navigator</title>
</head>
<body>
<% if (navigator.indexOf("MSIE") != -1)
{
%>
<p>Utilizati navigatorul Internet Explorer.</p>
<% } else { %>
<p> Utilizati navigatorul <%= navigator %>. </p>
<% } %>
</body>
</html>
```

Din acest exemplu se observă cum se poate intercala cod Java cu secvențe HTML. Instrucțiunea `if` conține cod HTML în cele două ramuri ale sale. De remarcat că secvențele Java și HTML alternează și nu sunt imbricate.

La începutul paginii JSP se declară o variabilă locală `navigator` și nu o dată membră.

Este recomandat a se utiliza cât mai puțin scriptlet-urile, deoarece acestea îmbină partea de design cu cea de programare, iar pentru cei care se ocupă de design ar fi de neînțeles pagina JSP. Alternativa ar fi crearea unei librării de taguri proprii.

11.3.9. Acțiuni

Acțiunile sunt taguri particulare predefinite. Acestea nu au corespondent un tag specific JSP (cele care încep cu `<%`), ci sunt numai taguri XML cu spațiul de nume `jsp`. Acțiunile conferă un nivel înalt de funcționalitate față de declarații, expresii și scriptlet-uri.

Există trei categorii de acțiuni standard:

1. cele utilizate pentru componente Bean;
2. cele pentru controlul din momentul execuției, cum ar fi redirectarea sau includerea;
3. cele care oferă suport pentru *plug-in*-uri Java.

11.3.9.1. Integrarea componentelor JavaBean

Tehnologia JSP oferă o integrare foarte bună a formularelor HTML cu componentele JavaBean. Pentru utilizarea unei componente JavaBean avem la dispoziție acțiunile din tabelul de mai jos:

Denumirea acțiunii (tagului)	Scopul acțiunii
jsp:useBean	Pregătește o componentă JavaBean pentru a fi utilizată în pagina JSP.
jsp:setProperty	Setează una sau mai multe proprietăți pentru o componentă JavaBean.
jsp:getProperty	Întoarce valoarea unei proprietăți a unei componente JavaBean sub forma unui String.

Acțiunea `jsp:useBean` se poate scrie ca tag de sine stătător (fără conținut):

```
<jsp:useBean id="nume" scope="scop" Specificare />
```

sau ca tag cu conținut:

```
<jsp:useBean id="nume" scope="scop" Specificare >
    corp_pentru_creare
</jsp:useBean>
```

Această acțiune duce la crearea unui obiect corespunzător componentei JavaBean și va avea numele `nume`.

Atributul `scope` indică care este durata de viață a componentei. Acesta poate avea valorile din tabelul următor:

scope	Durata existenței componentei
page	Componenta este disponibilă doar în pagina JSP curentă și va fi recreată la fiecare cerere.
request	Componenta este disponibilă pe tot parcursul cererii, inclusiv în paginile incluse sau redirectate.
session	Componenta este disponibilă pe tot parcursul sesiunii stabilite.
application	Componenta va fi disponibilă pentru toate sesiunile și își va înceta existența o dată cu aplicația Web.

Elementul `Specificare` este destul de flexibil și oferă o varietate de opțiuni care sunt prezentate în tabelul de mai jos:

Opțiuni	Semnificație
<code>class="numeClasa"</code>	Se specifică clasa corespunzătoare componentei (<code>numeClasa</code>).
<code>type="numeTip"</code> <code>class="numeClasa"</code>	Se indică tipul care se va utiliza în pagină pentru componentă (<code>numeTip</code>), care trebuie să fie compatibil cu tipul clasei. Se indică și numele clasei.

Opțiuni	Semnificație
type="numeTip" beanName="numeComp"	Se indică tipul care se va utiliza în pagină pentru componentă (numeTip), precum și numele componentei. Acesta trebuie să fie furnizat prin intermediul unei expresii JSP dat în forma JSP (cu simbolurile <%).
type="numeTip"	Se stabilește tipul componentei care va fi utilizat în cadrul paginii JSP.

Sintaxa acțiunii `jsp:setProperty` este următoarea:

```
<jsp:setProperty name="numeComp" extensie />
```

Numele componentei este cel stabilit de atributul `id` al acțiunii `jsp:useBean`. De exemplu, dacă avem inserată o componentă JavaBean astfel:

```
<jsp:useBean id="componental" ... />
```

atunci stabilirea (sau obținerea unei proprietăți) se realizează prin intermediul numelui `componental`:

```
<jsp:setProperty name="componental" ... />
```

Elementul `extensie` poate avea următoarele forme:

extensie	Descriere
property="numeProp"	Stabilește doar proprietatea indicată pentru parametrul cerut.
property="numeProp" param="numeParam"	Stabilește proprietatea specificată pentru parametrul indicat.
property="numeProp" value="valoare"	Setează proprietatea specificată cu valoarea dată. Valoarea trebuie să fie o expresie JSP care se evaluează în momentul execuției. Aceasta trebuie să fie în forma JSP (cu simbolurile <%).

Acțiunea `jsp:getProperty` este utilizată pentru obținerea valorii unei proprietăți. Aceasta are următoarea formă generală:

```
<jsp:getProperty name="numeComp" property="numeProprietate" />
```

În mod automat valoarea obținută este convertită la tipul `String`.

11.4. Concluzii

Având la bază tehnologia servleturilor, JSP-urile permit crearea de pagini Web dinamice. Sunt mult mai ușor de implementat decât servleturile și oferă posibilitatea de a crea aplicații Web complexe, putându-se utiliza componente JavaBeans, documente text sau binare, conexiuni la baze de date sau la diverse servere din rețea etc.

Ca și servleturile, se bucură de întregul suport oferit de limbajul Java.

11.5. Test grilă

Întrebarea 11.5.1. Care sunt obiectele predefinite pentru o pagină JSP?

- a) out
- b) session
- c) servlet
- d) exception

Întrebarea 11.5.2. Care metode ale unui JSP sunt apelate automat?

- a) `jspStart()`
- b) `jspInit()`
- c) `jspDestroy()`
- d) `jspFialize()`

Întrebarea 11.5.3. Care dintre afirmațiile următoare sunt adevărate?

- a) Doar o singură directivă poate exista într-o pagină JSP.
- b) O expresie nu poate conține o altă expresie.
- c) O expresie se termină cu simbolul punct și virgulă.
- d) Vizualizarea paginilor JSP se poate realiza și fără ajutorul unui server Web.

11.6. Exerciții propuse spre implementare

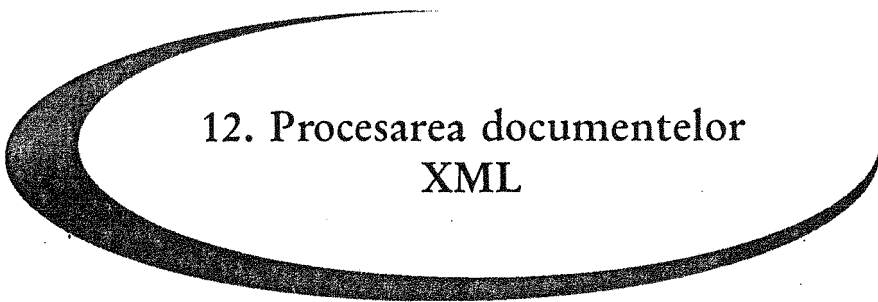
Exercițiul 11.6.1. Să se creeze o aplicație Web pentru rezervarea biletelor de călătorie pentru transportul feroviar.

Exercițiul 11.6.2. Să se construiască un sit cu anunțuri de mică publicitate. Orice utilizator poate adăuga un anunț. Acestea sunt grupate pe categorii și fiecare anunț are un anumit timp de valabilitate la expirarea căruia respectivul anunț va fi șters.

11.7. Proiecte propuse spre implementare

Proiectul 11.7.1. Să se elaboreze un sit de comerț electronic pentru un magazin universal. Acesta va cuprinde diverse categorii de produse, oferte promoționale, oferte de lichidări de stoc. Se va implementa un modul de căutare, coșul de cumpărături și un modul de administrare care se va ocupa de modificarea ofertelor, actualizarea informațiilor privitoare la produse etc. Datele vor fi memorate într-o bază de date.

Proiectul 11.7.2. Să se implementeze un sistem de chat online. Fiecare utilizator va avea un nume și o parolă, pot exista mai multe camere de discuții, se va da posibilitatea de a trimite mesaje private sau publice, de a vizualiza lista persoanelor din camera curentă, obținerea de informații despre un anumit utilizator etc.



12. Procesarea documentelor XML

12.1. Cuvinte cheie

- documente XML
- XML, SAX, DOM, XSLT
- spații de nume XML, validare documente XML, DTD
- transformări, documente XSL

12.2. Standarde și specificații

XML (eXtensible Markup Language) este un meta-limbaj definit de Consorțiul Web (W3C) și poate fi utilizat pentru a descrie o varietate de limbaje de marcare ierarhice. Este un set de reguli, specificații și convenții pentru structurarea datelor în fișiere text (eng. *plain text*). Avantajul utilizării unui fișier text față de unul binar constă în faptul că programatorul sau utilizatorul poate citi, înțelege și utiliza respectivul fișier fără a fi nevoie de programul care l-a produs. Oricum, principalii producători și utilizatori de date XML sunt programele de calculator, și nu utilizatorii.

Ca și HTML (sau mai nou XHTML), XML utilizează taguri și atribute. Tagurile sunt cuvintele cuprinse între caracterele „<” și „>”, iar atributele sunt secvențe de forma `nume="valoare"` și apar în interiorul tagurilor. Pe când HTML-ul specifică semnificația fiecărui tag și atribut (adică modul de vizualizare în navigatoarele Web), XML-ul folosește taguri doar pentru a delimita datele, iar interpretarea este dată de aplicația care îl utilizează. Altfel spus, XML definește doar structura documentului, nu și semantica acestuia.

Dezvoltarea standardului XML a fost începută în 1996 de către Consorțiul Web, iar în 1998 a apărut prima recomandare, revizuită în octombrie 2000. Tehnologia nu este cu totul nouă, deoarece se bazează pe SGML (Standard Generalized Markup Language), care a apărut în anii 1980, iar în 1986 a devenit standard ISO. Creatorii XML-ului au luat părțile cele mai bune de la SGML și au construit o tehnologie la fel de puternică ca și SGML, dar mult mai simplă și mai ușor de utilizat.

W3C XML 1.0 Recommendation (Second edition) stabilește sintaxa XML. Detalii despre grupul de lucru pentru XML a Consorțiului Web se găsesc la <http://www.w3c.org/XML/>. Recomandarea XML 1.0 a Consorțiului Web (a doua ediție) este disponibilă la <http://www.w3c.org/TR/2000/REC-xml-20001006>.

W3C XML Namespaces 1.0 Recommendation definește sintaxa și semantica structurilor XML pentru a nu apărea conflicte între elementele cu același nume. Recomandarea *spațiilor de nume* (eng. *namespaces*) pentru XML poate fi accesată la adresa <http://www.w3.org/TR/1999/REC-xml-names-19990114/>.

Simple API for XML Parsing (SAX) 2.0 oferă o interfață bazată pe evenimente pentru parsarea documentelor XML. Detalii despre SAX 2.0 se găsesc la <http://www.megginson.com/SAX/index.html>, iar extensiile SAX 2.0 sunt la <http://www.megginson.com/Software/sax2-ext-1.0.zip>. În secțiunea 12.4. este prezentată modalitatea de procesare a documentelor XML utilizând SAX.

Document Object Model (DOM) Level 2 oferă un set de interfețe definite de W3C DOM Working Group (Grupul de lucru DOM a W3C). Acestea descriu facilitățile pentru reprezentarea arborescentă a documentelor XML (sau HTML) în aplicații. Specificațiile acestor interfețe sunt definite prin intermediul limbajului

Interface Definition Language (IDL) – limbaj pentru definirea interfețelor –, care este independent de limbajul de programare. La adresa <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/> se găsesc specificațiile DOM nivelul 2. Subcapitolul 12.5. cuprinde suportul oferit de limbajul Java pentru construirea arborelui DOM corespunzător unui document XML și modalitățile de parcurgere a acestuia.

XSLT 1.0 (XSL Transformations) descrie un limbaj pentru transformarea documentelor XML în alte documente XML sau text. Acest limbaj este definit de Consorțiul Web, grupul de lucru pentru XSL (eXtensible Stylesheet Language), iar specificațiile sunt disponibile la <http://www.w3.org/TR/1999/REC-xslt-19991116>. (De fapt XSL este alcătuit din XSLT și XPath).

12.3. Instalare

Pachetele necesare procesării documentelor XML sunt incluse în JDK începând cu versiunea 1.4. Eventual, aceste pachete se pot procura separat și instala urmând indicațiile care însoțesc respectivele pachete.

12.4. SAX

Simple API for XML Parsing (SAX) permite procesarea documentelor XML prin intermediul unor evenimente. Modalitatea de procesare este foarte simplă: se parcurge fișierul respectiv, iar la apariția elementelor XML se vor lansa evenimente care pot fi tratate. Programatorul va trata evenimentele dorite și în funcție de elementul (tagul XML) generator va realiza anumite operații.

12.4.1. Pachete necesare

Pachetele necesare procesării documentelor XML prin intermediul SAX sunt următoarele:

- `javax.xml.parsers`
Conține clasele necesare procesării documentelor XML (atât cu SAX, cât și cu DOM).
- `org.xml.sax`
Ofere interfețele necesare procesării documentelor XML prin SAX.
- `org.xml.sax.helpers`
Definește clase ajutoare pentru utilizarea interfețelor SAX.
- `org.xml.sax.ext`
Ofere două interfețe pentru a extinde SAX-ul.

12.4.2. Exemplu de document XML

Pentru studiul procesării documentelor XML, vom considera fișierul `optionale.xml`, care cuprinde informații referitoare la trei cursuri opționale:

```
<?xml version="1.0" ?>
<optionale>
  <optional id="C003">
    <nume>Programarea interfetelor utilizator</nume>
    <profesor>Sabin Corneliu Buraga</profesor>
    <an>3</an>
    <url>http://www.infoiasi.ro/~busaco/courses/piu/</url>
    <studenti min="15" max="60" />
    <recomandari>
      <curs>Birotica</curs>
      <curs>Grafica</curs>
    </recomandari>
  </optional>

  <optional id="C014">
    <nume>Compilare paralela</nume>
    <profesor>Stefan Andrei</profesor>
    <an>4</an>
    <url>http://www.infoiasi.ro/~stefan/compil/</url>
    <studenti min="15" max="80" />
    <recomandari>
      <curs>Tehnici de compilare</curs>
      <curs>Programare Java</curs>
    </recomandari>
  </optional>

  <optional id="C009">
    <nume>Medii virtuale</nume>
    <profesor>Stefan Ciprian Tanasa</profesor>
    <an>4</an>
    <url>http://www.infoiasi.ro/~stanasa/vr/</url>
    <studenti min="10" max="100" />
    <recomandari>
      <curs>Geometrie computationala</curs>
      <curs>Programare Java</curs>
    </recomandari>
  </optional>
</optionale>
```

12.4.3. Obținerea informațiilor din documentele XML

În Java, fiecărui eveniment îi va corespunde o metodă definită în interfața `ContentHandler`. Metodele sunt apelate automat de către *parser*, iar cele mai importante sunt următoarele:

- `void startDocument()`
Este apelată automat în momentul în care se începe procesarea documentului XML.
- `void endDocument()`
Se execută la terminarea procesării documentului.
- `void startElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName, Attributes atts)`
Se apelează la apariția unui tag. Adresa spațiului de nume va fi transmisă în parametrul `namespaceURI` (dacă spațiul de nume are atașat un URI și dacă este activată facilitățile de procesare a spațiilor de nume). În cazul în care facilitățile de procesare a spațiilor de nume este activă, atunci `localName` va conține numele tagului fără prefix, iar `qName` cu tot cu prefix, altfel `localName` va conține șirul vid, iar `qName` numele tagului. Atributele vor fi memorate în obiectul `atts`. Dacă tagul nu posedă atribute, atunci `atts` va fi un obiect vid.
- `void endElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName)`
Această metodă este invocată atunci când este întâlnit un tag de sfârșit, iar parametrii au aceeași semnificație ca la metoda precedentă. Dacă tagul este de sine stătător (de exemplu: `<studenti />`), atunci metoda `endElement` este apelată imediat după `startElement`.
- `void characters(char[] ch, int start, int length)`
Se execută la apariția unui text aflat între taguri. Respectivul text va fi regăsit în tabloul `ch` începând cu poziția `start` și va conține `length` caractere.
- `void processingInstruction(java.lang.String target, java.lang.String data)`
Metoda va fi invocată la apariția unei instrucțiuni de procesare. Parametrul `target` va conține numele instrucțiunii de procesare, iar `data`, informațiile aferente. De exemplu, dacă în document există instrucțiunea `<?java document="Sax.java" ?>`, atunci în `target` se va regăsi cuvântul `java`, iar `data` va avea valoarea `document="Sax.java"`.

Toate metodele din interfața `ContentHandler` transmit mai departe excepția `SAXException`.

Interfața `ContentHandler` este implementată de clasa `DefaultHandler`. Aceasta mai implementează interfețele `DTDHandler`, `EntityHandler` și `ErrorHandler`. Implicit, fiecare metodă corespunzătoare interfețelor amintite nu va avea vreun efect.

Pentru construirea unui obiect pentru efectuarea operațiilor dorite, care se vor executa la procesarea unui document XML, va trebui să declarăm o clasă care să implementeze interfața `ContentHandler` sau să extindă clasa `DefaultHandler`.

Clasa `SAXParserFactory` este utilizată pentru crearea și configurarea de parsere bazate pe SAX. Nu este recomandat să se utilizeze mai multe instanțe ale acestei clase într-un fir de execuție. De altfel, trebuie avută mare grijă în utilizarea unui astfel de obiect de către mai multe fire de execuție. O aplicație poate obține mai multe instanțe ale clasei `SAXParser` dintr-un obiect de tip `SAXParserFactory`. Pentru obținerea unei instanțe a clasei `SAXParserFactory` se utilizează metoda statică `newInstance()` din aceeași clasă.

Clasa `SAXParser` este utilizată pentru procesarea efectivă a documentelor XML. Instanțele acestei clase se obțin prin apelul metodei `newInstance()` din clasa `SAXParserFactory`. Nu este indicată utilizarea concomitentă a unui obiect de tip `SAXParser` de către mai multe fire de execuție. Fișierul XML poate fi dat ca un obiect de tip `File`, `InputStream`, `InputStream` sau `String` (URI). Acesta este transmis ca parametru metodei `parse()` împreună cu un obiect `DefaultHandler`.

Exemplul 12.4.1. În fișierul `Sax01.java` am definit clasa `MySAXHandler` care extinde clasa `DefaultHandler`. Clasa `MySAXHandler` este utilizată pentru procesarea fișierului `optionale.xml`. Clasa `Sax01` va conține funcția `main()`, care va crea obiectele necesare și va iniția parsarea documentului XML.

Programul va avea ca efect afișarea tuturor cursurilor opționale. Fișierul `Sax01.java` este următorul:

```
import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;

class MySAXHandler extends DefaultHandler {
    // indicator pentru a stabili care informatie cuprinsa
    // intre taguri va fi afisata pe ecran
    private int flag;

    // constructor implicit
    public MySAXHandler() { flag=0; }

    // se apeleaza la inceputul parsarii documentului
    public void startDocument() throws SAXException {
        System.out.println("Cursurile optionale sunt:");
    }

    // se apeleaza cand se ajunge la un tag XML
    public void startElement(String ns, String local, String
        name, Attributes atts)
        throws SAXException
    {
        // ne intereseaza doar tagurile nume
```

```
    if (name.equals("nume")) flag=1;
}

// se apeleaza cand se intalneste un tag de sfarsit
public void endElement(String ns, String local, String name)
    throws SAXException
{
    if (name.equals("nume")) flag=0;
}

// se apeleaza cand se ajunge la o informatie dintre taguri
public void characters(char[] ch, int start, int length)
    throws SAXException
{
    if (flag==1)
    {
        System.out.print(" * ");
        for(int i=0;i<length;i++)
            System.out.print(ch[start+i]);
        System.out.println("");
    }
}

public class Sax01 {
    public static void main(String arg[])
    {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        DefaultHandler handler = new MySAXHandler();
        SAXParser parser;

        try {
            parser = factory.newSAXParser();
            parser.parse("optionale.xml", handler);
        } catch (SAXException se) {
            System.out.println("SAXException:\n" + se.getMessage());
        } catch (IOException ioe) {
            System.out.println("IOException:\n" + ioe.getMessage());
        } catch (ParserConfigurationException pce) {
            System.out.println("ParserConfigurationException:\n" +
                pce.getMessage());
        }
    }
}
```

Se observă că la începutul procesării documentului se va afișa mesajul "Cursurile optionale sunt:". Apoi se vor considera doar tagurile nume care vor seta indicatorul flag. Când se va întâlni tagul de sfârșit, se va atribui indicatorului valoarea zero. Șirurile de caractere din afara tagurilor vor fi afișate doar dacă a fost setat indicatorul, adică se află în interiorul tagului nume.

Programul are ca efect afișarea următorului text:

Cursurile optionale sunt:

- * Programarea interfetelor utilizator
- * Compilare paralela
- * Medii virtuale

12.4.4. Obținerea valorilor atributelor unui tag

Dacă dorim să obținem pentru fiecare curs numărul minim, respectiv maxim de studenți care pot urma cursul, atunci va trebui să preluăm valorile atributelor min și max ale tagului studenti. Acest lucru îl obținem prin adăugarea la exemplul 12.4.1. a următorului cod înainte de sfârșitul metodei startElement() din clasa MySAXHandler:

```
else if (name.equals("studenti"))
{
    String min=atts.getValue("min");
    String max=atts.getValue("max");
    if (min!=null) // daca exista atributul min
        System.out.println("\tNumarul minim de studenti: " + min);
    if (max!=null) // daca exista atributul max
        System.out.println("\tNumarul maxim de studenti: " + max);
}
```

Se testează mai întâi dacă s-a ajuns la tagul studenti, iar în caz afirmativ, se vor prelua valorile acestora prin apelul metodei getValue() din interfața Attributes. Acesta primește ca parametru numele atributului pentru care se dorește aflarea valorii. În cazul în care atributul nu a fost specificat, atunci se întoarce valoarea null.

Pentru aflarea tuturor atributelor se poate proceda în felul următor: se va obține mai întâi numărul total de atribute ale tagului curent prin utilizarea metodei getLength(), apoi prin intermediul unei structuri repetitive (for, while) se vor parcurge indecșii; pentru aflarea valorii unui atribut care are un anumit index, se va utiliza metoda getValue(), iar pentru numele atributului, getQName().

De exemplu, pentru afișarea tuturor atributelor tagului studenti cu valorile corespunzătoare, se va înlocui codul adăugat anterior cu următorul:

```
else if (name.equals("studenti"))
{
    for (int i=0; i<atts.getLength(); i++)
```

```
System.out.println("\t" + atts.getQName(i) + "=" + atts.
    getValue(i));
}
```

Rularea programului rezultat va avea ca efect:

Cursurile optionale sunt:

- * Programarea interfetelor utilizator
 - min=15
 - max=60
- * Compilare paralela
 - min=15
 - max=80
- * Medii virtuale
 - min=10
 - max=100

12.4.5. Procesarea documentelor XML care conțin spații de nume

Apar situații în care documentele XML utilizează spațiile de nume (*namespace*). Acest lucru este util pentru a specifica contextul anumitor elemente XML. Într-un document XML, un spațiu de nume este utilizat prin intermediul atributului xmlns, care este eventual urmat de carcterul „:” și apoi de numele unui prefix. Dacă nu se specifică nici un prefix, atunci este considerat ca fiind vid. Acest prefix va putea fi utilizat în definirea tagurilor din interiorul tagului care conține atributul xmlns. De exemplu, pentru a indica că un tag face parte dintr-un spațiu de nume pentru care am stabilit prefixul „prefix”, vom utiliza construcția prefix:tag în loc de denumirea tagului (tag). Mai multe detalii referitoare la spațiile de nume se pot obține din specificațiile consorțiului Web la <http://www.w3.org/TR/1999/REC-xml-names-19990114/>.

Vom considera ca exemplu documentul optionale_ns.xml, acesta obținându-se prin adăugarea unor spații de nume la documentul optionale.xml definit în secțiunea 12.4.2:

```
<?xml version="1.0" ?>
<optionale xmlns="urn:infoiasi.ro:Cursuri">
<optional id="C003">
    <nume>Programarea interfetelor utilizator</nume>
    <profesor>Sabin Corneliu Buraga</profesor>
    <an>3</an>
    <url>http://www.infoiasi.ro/~busaco/courses/piu</url>
    <studenti min="15" max="60" />
    <recomandari>
        <curs>Birotica</curs>
        <curs>Grafica</curs>
```

```

</recomandari>
</optional>

<optional id="C014" xmlns:Prof="urn:infoiasi.ro:Profesori">
  <nume>Compilare paralela</nume>
  <Profesori:profesor>Stefan Andrei</Profesori:profesor>
  <an>4</an>
  <url>http://www.infoiasi.ro/~stefan/compil</url>
  <studenti min="15" max="80" />
  <recomandari>
    <curs>Tehnici de compilare</curs>
    <curs>Programare Java</curs>
  </recomandari>
</optional>

<optional id="C009" xmlns="urn:infoiasi.ro:Optional">
  <nume>Medii virtuale</nume>
  <profesor>Stefan Ciprian Tanasa</profesor>
  <an>4</an>
  <url>http://www.infoiasi.ro/~stanasa/vr</url>
  <studenti min="10" max="100" />
  <recomandari>
    <curs>Geometrie computationala</curs>
    <curs>Programare Java</curs>
  </recomandari>
</optional>
</optionale>

```

Pentru a indica faptul că documentul care urmează a fi procesat conține declarații ale spațiilor de nume, se va invoca pentru un obiect de tip SAXParserFactory metoda setNamespaceAware() cu valoarea true pentru singurul său parametru.

Exemplul 12.4.2. Programul următor ilustrează modalitatea de procesare a informațiilor referitoare la spațiile de nume:

```

import javax.xml.parsers.*;
import org.xml.sax.helpers.*;
import org.xml.sax.*;
import java.io.*;

class MySAXHandler extends DefaultHandler {

    // se apeleaza cand se ajunge la un tag XML
    public void startElement(String ns, String local, String
        name, Attributes atts)

```

```

    throws SAXException
    {
        // ne intereseaza doar tagul profesor
        if (local.equals("profesor"))
            System.out.println("NS: "+ns+"\nName: "+name);
    }

    // se apeleaza cand se intalneste o specificare a unui spatiu
    // de nume
    public void startPrefixMapping(String prefix, String uri)
        throws SAXException
    {
        System.out.println(">> Prefix: "+prefix+"\tURI: "+uri);
    }

    // se invoca atunci cand se iese din domeniul
    // de valabilitate a respectivului spatiu de nume
    public void endPrefixMapping(String prefix)
        throws SAXException
    {
        System.out.println(">> End Prefix: "+prefix);
    }
}

public class Sax03bis {
    public static void main(String arg[])
    {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        DefaultHandler handler = new MySAXHandler();
        SAXParser parser;

        // se invoca procesarea spatiilor de nume
        factory.setNamespaceAware(true);

        try {
            parser = factory.newSAXParser();
            parser.parse("optionale_ns.xml", handler);
        } catch (SAXException se) {
            System.out.println("SAXException:\n" +
                se.getMessage());
        } catch (IOException ioe) {
            System.out.println("IOException:\n" + ioe.getMessage());
        } catch (ParserConfigurationException pce) {
            System.out.println("ParserConfigurationException:\n" +

```



```

try {
    parser = factory.newSAXParser();
    xmlReader = parser.getXMLReader();
    xmlReader.setContentHandler(handler);
    xmlReader.parse("optionale.xml");
} catch (SAXException se) {
    System.out.println("SAXException:\n" + se.getMessage());
} catch (IOException ioe) {
    System.out.println("IOException:\n" + ioe.getMessage());
} catch (ParserConfigurationException pce) {
    System.out.println("ParserConfigurationException:\n" +
        pce.getMessage());
}

```

Clasa MySAXHandler este aceeași ca cea din exemplul 12.4.2. Vom utiliza interfața XMLReader pentru a realiza parsarea documentului XML. Obținem o instanță în urma apelului metodei getXMLReader() din clasa SAXParser.

Pentru stabilirea obiectului care implementează interfața ContentHandler se va utiliza metoda setContentHandler(). Pentru pornirea parsării se va apela metoda parse(), care are ca parametru un șir de caractere (String) ce semnifică identificatorul unic al fișierului (URI).

De asemenea, pentru obiectul factory am apelat metoda setValidating() care activează validarea documentului XML.

Rularea programului obținut va genera un mesaj de eroare care semnalează faptul că tagul optional este incomplet, deoarece este obligatorie indicarea celorlalte subtaguri. Deci acum programul cercetează dacă documentul XML respectă structura specificată. În plus, mai este afișat un mesaj de avertizare care ne anunță că nu a fost specificat un obiect care să trateze erorile și că este utilizat unul implicit care nu reține decât primele 10 erori.

Trebuie să definim un obiect a cărui clasă să implementeze interfața ErrorHandler. Putem să specificăm obiectul handler, care este o instanță a clasei MySAXHandler, iar aceasta extinde clasa DefaultHandler. Aceasta implementează și interfața ErrorHandler. În funcția main(), înainte de parsare vom adăuga următoarea linie de cod:

```
xmlReader.setErrorHandler(handler);
```

Această modificare va face să nu mai apară nici un mesaj de eroare, deoarece clasa DefaultHandler implementează funcțiile specificate în interfața ErrorHandler ca funcții vide (fără nici o instrucțiune, deci fără nici un efect). Pentru a semnaliza erorile ce apar, va trebui să redefinim în clasa MySAXHandler metodele warning(), error() și fatalError(), astfel:

```

public void warning(SAXParseException spe) throws SAXException {
    System.out.println("Warning: " + spe.getMessage());
}

```

```

public void error(SAXParseException spe) throws SAXException {
    String message = "Error: " + spe.getMessage();
    throw new SAXException(message);
}

```

```

public void fatalError(SAXParseException spe) throws
    SAXException {
    String message = "Fatal Error: " + spe.getMessage();
    throw new SAXException(message);
}

```

În caz de eroare se va arunca o excepție SAXException cu un mesaj corespunzător. Acum execuția programului va afișa următorul mesaj de eroare:

```

SAXException:
Error: The content of element type "optional" is incomplete, it
must match "(nume,profesor,an,url,studenti,recomandari)".

```

Astfel, se poate realiza validarea documentelor XML pentru care s-a specificat structura sa prin intermediul unui DTD.

Observație: Dacă parserul are activată validarea documentului XML și acesta nu are definit un DTD, atunci se va genera un mesaj de atenționare, iar apoi se va arunca o excepție:

```

Warning: Valid documents must have a <!DOCTYPE declaration.
SAXException: Error: Element type "optionale" is not declared.

```

12.5. DOM

Document Object Model (DOM) este un standard stabilit de consorțiul Web și conține interfețe pentru manipularea arborelui de obiecte asociat unui document XML. Cu ajutorul DOM-ului se pot adăuga noi noduri în arbore (noi taguri, atribute), respectiv modifica și șterge cele existente.

De exemplu, dacă avem documentul XML de mai jos:

```

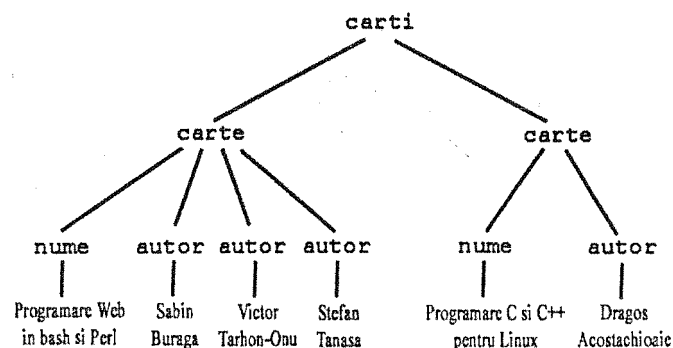
<?xml version="1.0" ?>
<carti>
  <carte>
    <nume>Programare Web in bash si Perl</nume>
    <autor>Sabin Buraga</autor>
    <autor>Victor Tarhon-Onu</autor>
    <autor>Stefan Tanasa</autor>
  
```

```

</carte>
<carte>
  <nume>Programare C si C++ pentru Linux</nume>
  <autor>Dragos Acostachioaie</autor>
</carte>
</carti>

```

atunci arborele asociat documentului XML va fi:



Atributele unui tag vor fi considerate ca informații atașate nodului corespunzător în arborele DOM.

12.5.1. Pachete necesare

Pachetele necesare procesării documentelor XML prin intermediul DOM sunt următoarele:

- `javax.xml.parsers`
Conține clasele necesare procesării documentelor XML (atât cu SAX, cât și cu DOM).
- `org.w3c.org`
Oferă interfețele utilizate pentru reprezentarea arborescentă a documentelor XML.

12.5.2. Crearea arborelui asociat unui document XML

Înainte de crearea unui arbore trebuie să obținem o instanță a unui obiect de tip `DocumentBuilderFactory`. Acesta este util pentru obținerea de obiecte `DocumentBuilder` care, la rândul lor, permit crearea arborilor asociați documentelor XML.

Obținerea unei instanțe a clasei `DocumentBuilderFactory` se realizează în urma apelului metodei statice `newInstance()` din aceeași clasă. Metoda `newDocumentBuilder()` întoarce o referință la un obiect de tip `DocumentBuilder` care posedă metoda `parse()`. Aceasta este utilizată la parsarea documentului XML și la crearea arborelui corespunzător.

Pentru construirea arborelui corespunzător documentului `optionale.xml` vom utiliza următorul cod:

```

DocumentBuilderFactory factory = DocumentBuilderFactory.
    newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse("optionale.xml");

```

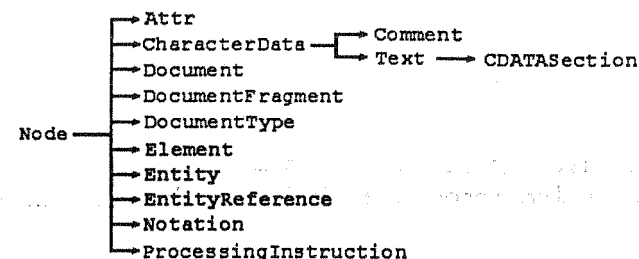
În memorie s-a construit arborele corespunzător documentului `optionale.xml`, iar referirea la acesta se va realiza prin intermediul interfeței `Document`.

Nodul rădăcină este returnat de metoda `getDocumentElement()` din interfața `Document`.

12.5.3. Manipularea arborelui DOM

12.5.3.1. Interfața Node

Fiecărui nod din arbore îi corespunde un obiect care implementează interfața `Node`. Aceasta posedă subinterfețe corespunzătoare fiecărui tip de nod. Spre exemplu, unui tag îi va corespunde interfața `Element`, pentru un comentariu `Comment` ș.a.m.d. Ierarhia de interfețe corespunzătoare nodurilor arborelui este descrisă în figura de mai jos:



Pentru identificarea tipului nodului avem la dispoziție metoda `getNodeType()` din interfața `Node`. Aceasta returnează o valoare de tip `short` care se regăsește printre membrii statici din respectiva interfață:

Data membră	Interfața corespunzătoare	Explicație
<code>ATTRIBUTE_NODE</code>	<code>Attr</code>	Nod de tip atribut
<code>CDATA_SECTION_NODE</code>	<code>CDATASection</code>	Nod corespunzător unei secțiuni CDATA
<code>COMMENT_NODE</code>	<code>Comment</code>	Nod corespunzător unui comentariu
<code>DOCUMENT_FRAGMENT_NODE</code>	<code>DocumentFragment</code>	Nod corespunzător unui fragment de document
<code>DOCUMENT_NODE</code>	<code>Document</code>	Nod corespunzător unui document XML

Data membră	Interfața corespunzătoare	Explicație
DOCUMENT_TYPE_NODE	DocumentType	Nod corespunzător unei secțiuni DOCTYPE
ELEMENT_NODE	Element	Nod corespunzător unui tag
ENTITY_NODE	Entity	Nod corespunzător unei entități
ENTITY_REFERENCE_NODE	EntityReference	Nod corespunzător unei referințe
NOTATION_NODE	Notation	Nod corespunzător unei notații
PROCESSING_INSTRUCTION_NODE	ProcessingInstruction	Nod corespunzător unei instrucțiuni de procesare
TEXT_NODE	Text	Nod de tip text

Fiecare nod are o listă cu nodurile descendente și una cu atribute. Eventual, acestea pot fi vide.

Cele mai importante metode din clasa Node sunt:

- `boolean hasChildNodes()`
Testează dacă nodul respectiv are descendenți.
- `boolean hasAttributes()`
Testează dacă nodul în cauză posedă atribute.
- `NodeList getChildNodes()`
Returnează lista nodurilor descendente.
- `Node getFirstChild()`
Returnează primul nod descendent.
- `Node getLastChild()`
Returnează ultimul nod descendent.
- `Node getNextSibling()`
Întoarce nodul următor celui curent din lista nodurilor descendente ale nodului părinte.
- `Node getPreviousSibling()`
Întoarce nodul precedent celui curent din lista nodurilor descendente ale nodului părinte.
- `Node getParentNode()`
Returnează nodul părinte nodului în cauză.
- `Node appendChild(Node newChild)`
Adaugă nodul specificat la sfârșitul listei nodurilor descendente.
- `Node cloneNode(boolean deep)`
Întoarce o copie a nodului curent (se realizează și copierea atributelor, dacă acestea există). Dacă parametrul are valoarea `true`, atunci se realizează o copie și pentru tot subarborul nodului respectiv.
- `Node insertBefore(Node newChild, Node refChild)`
Inserează un nod nou (`newChild`) înaintea unui nod specificat (`refChild`) în lista nodurilor descendente. Returnează nodul inserat.
- `Node removeChild(Node oldChild)`

Elimină din lista nodurilor descendente nodul specificat. Se returnează nodul eliminat.

- `Node replaceChild(Node newChild, Node oldChild)`
Se înlocuiește nodul `oldChild` cu `newChild` în lista nodurilor descendente ale nodului curent. Se returnează nodul care a fost înlocuit.
- `NamedNodeMap getAttributes()`
Returnează o colecție cu atributele corespunzătoare nodului curent, dacă acesta este de tip `Element` sau `null`, în caz contrar. Observăm că doar nodurile corespunzătoare tagurilor posedă o colecție de atribute.
- `Document getOwnerDocument()`
Furnizează un obiect de tip `Document` corespunzător nodului curent. Orice nod împreună cu nodurile descendenților săi formează un arbore.
- `void normalize()`
Unește nodurile de tip `Text` adiacente și le elimină pe cele vide.
- `void setNodeValue(String newValue)`
Modifică valoarea memorată în nodul curent, în funcție de natura acestuia.

Valorile returnate de metodele `getNodeName()` și `getNodeValue()` depind de natura nodurilor, după cum urmează:

Interfața	<code>getNodeName()</code>	<code>getNodeValue()</code>
Attr	numele atributului	valoarea atributului
CDATASection	"#cdata-section"	conținutul secțiunii CDATA
Comment	"#comment"	conținutul comentariului
Document	"#document"	null
DocumentFragment	"#document-fragment"	null
DocumentType	numele tipului de document	null
Element	numele tagului	null
Entity	numele entității	null
EntityReference	numele entității referite	null
Notation	numele notației	null
ProcessingInstruction	șinta (eng. <i>target</i>)	conținutul (mai puțin șinta)
Text	"#text"	textul conținut

Exemplul 12.5.1. Programul Java de mai jos procesează documentul `optionale.xml` (conținutul acestuia este dat în secțiunea 12.4.2.) și afișează lista cursurilor opționale.

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import java.io.*;

public class DOM01 {
```

```

public static void main(String args[])
{
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.
            newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();

        /* Obținerea arborelui asociat */
        Document document = builder.parse("optionale.xml");

        /* Obținerea elementului radacina */
        Element elt = document.getDocumentElement();
        Node opt, n;
        System.out.println("Elementul radacina este: "
            +elt.getTagName());

        /* Parcurgerea arborelui */
        System.out.println("Cursurile optionale sunt:");

        /* Daca elementul radacina nu contine elemente afisez un
        mesaj de eroare*/
        if (!elt.hasChildNodes())
        {
            System.err.println("Radacina trebuie sa contina
            subelemente!");
            System.exit(1);
        }

        /* Obțin nodul corespunzător primului optional */
        opt = elt.getFirstChild();

        /* Cat timp mai am optionale neprocesate */
        while(opt != null)
        {
            /* Parcurgerea nodurilor descendente */
            n = opt.getFirstChild();
            while(n != null)
            /* Daca am ajuns la nodul
            corespunzător tagului nume */
            if (n.getNodeName().equals("nume"))
            {
                n = n.getFirstChild();
                System.out.print(" * ");
                while(n != null)
                {

```

```

                /* Afisez textul din interiorul tagului nume*/
                if (n.getNodeType() == Node.TEXT_NODE)
                    System.out.println(n.getNodeValue().trim());
                /* trec la urmatorul nod din
                lista nodurilor descendente */
                n = n.getNextSibling();
            }
            n=null; /* iesire fortata din while */
        }
        else
            n = n.getNextSibling();
        /* trecerea la urmatorul optional */
        opt = opt.getNextSibling();
    }

    /* Prinderea exceptiilor */
} catch (Exception e) {
    System.out.println("Exceptie: "+e.getMessage());
}
}
}

```

În exemplul de mai sus ne-am folosit de faptul că știam dinainte structura documentului XML. Acest lucru ne-a permis să adoptăm o soluție iterativă a problemei.

Pentru obținerea informațiilor referitoare la nodurile arborilor sunt utilizate interfețele `NamedNodeMap` și `NodeList`.

Interfața `NamedNodeMap` este definită pentru a reprezenta o colecție de noduri care pot fi referite cu ajutorul numerelor acestora. Elementele pot fi accesate și prin intermediul indecșilor. Este permisă eliminarea și înlocuirea anumitor noduri. Metodele acestei interfețe sunt prezentate în tabelul următor:

Prototipul metodei	Descrierea metodei
<code>int getLength()</code>	Întoarce numărul de elemente din colecție.
<code>Node item(int index)</code>	Returnează nodul cu indexul specificat.
<code>Node getNamedItem(String nume)</code>	Returnează nodul cu numele specificat sau null în cazul în care acesta nu există.
<code>Node setNamedItem(Node nod) throws DOMException</code>	Adaugă nodul specificat la colecție. Dacă mai există un nod cu același nume, atunci acesta va fi înlocuit. Se returnează nodul înlocuit sau null în cazul în care nu s-a realizat doar o adăugare.
<code>Node removeNamedItem(String nume) throws DOMException</code>	Elimină nodul cu numele specificat din colecție. Se returnează nodul eliminat, iar în cazul în care acesta nu există în colecție, se aruncă o excepție <code>DOMException</code> .
<code>Node getNamedItemNS(String URI, String numeLocal)</code>	Întoarce nodul cu numele local specificat aflat în spațiul de nume cu URI-ul indicat sau null, dacă acesta nu este găsit.

Prototipul metodei	Descrierea metodei
Node setNamedItemNS (Node nod) throws DOMException	Are același efect ca <code>setNamedItem()</code> , în plus specificându-se spațiul de nume în nodul specificat.
Node removeNamedItemNS (String URI, String numeLocal) throws DOMException	Se elimină nodul cu numele și URI-ul spațiului de nume specificat.

Interfața `NodeList` reprezintă o colecție ordonată de noduri. Se oferă doar obținerea nodurilor din colecție, nu și modificarea acestora. Cele două metode ale interfeței `NodeList` sunt următoarele:

Prototipul metodei	Descrierea metodei
int getLength ()	Furnizează numărul de elemente din colecție.
Node item (int index)	Returnează nodul cu indexul specificat. Dacă indexul este invalid, atunci se întoarce valoarea <code>null</code> .

Deși metodele interfeței `NodeList` se regăsesc printre cele din `NamedNodeMap`, nu există nici o relație de moștenire între cele două interfețe.

Exemplul 12.5.2. Programul următor va afișa valorile tuturor atributelor din documentul XML, iar maniera de parcurgere a arborelui este una recursivă. Sunt utilizate interfețele `NamedNodeMap` și `NodeList` pentru parcurgerea atributelor unui nod de tip `Element`, respectiv pentru parcurgerea nodurilor descendente.

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import java.io.*;

class UtilClass {
    public void showAttributesRec(Node n)
    {
        NodeList aux;
        NamedNodeMap map;

        /* Dacă tagul are attribute atunci le vom afișa */
        if (n.hasAttributes())
        {
            System.out.println("Atributele tagului "
                + n.getNodeName() + " sunt: ");
            /* obținem lista atributelor */
            map = n.getAttributes();
            /* parcurgem întreaga lista de attribute */
            for(int i=0; i<map.getLength(); i++)
                System.out.println("\t"+map.item(i).getNodeName() +
```

```
"+"\t"+map.item(i).getNodeValue()+"\n");
        }
        /* dacă avem noduri descendente */
        if (n.hasChildNodes())
        {
            aux = n.getChildNodes();
            /* parcurgem lista nodurilor descendente */
            for(int i=0; i<aux.getLength(); i++)
                /* apel recursiv */
                showAttributesRec(aux.item(i));
        }
    }
}

public class DOM02 {
    public static void main(String args[])
    {
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.
                newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse("optionale.xml");
            Element elt = document.getDocumentElement();

            /* Parcurgerea arborelui */
            UtilClass util = new UtilClass();
            util.showAttributesRec(elt);

            /* Prinderea excepțiilor */
        } catch (Exception e) {
            System.out.println("Excepție: "+e.getMessage());
        }
    }
}
```

Am utilizat o clasă ajutătoare (`UtilClass`) în care am definit o metodă `showAttributesRec()` pentru parcurgerea recursivă a arborelui unui document. Dacă un nod posedă attribute, atunci acesta este de tip `Element` (cu siguranță este un tag), iar metoda `getNodeName()` va returna numele tagului.

În cazul în care nodul posedă descendenți (alte taguri), atunci se va apela metoda `showAttributesRec()` și pentru aceste noduri. Se observă faptul că nu se ține cont de numele tagului pentru care se afișează attributele, de aceea programul poate procesa orice tip de document XML.

La execuția programului anterior se va afișa la consolă:

Elementul radacina este: optionale
 Atributele tagului optional sunt:
 id="C003"
 Atributele tagului studenti sunt:
 min="15"
 max="60"
 Atributele tagului optional sunt:
 id="C014"
 Atributele tagului studenti sunt:
 min="15"
 max="80"
 Atributele tagului optional sunt:
 id="C009"
 Atributele tagului studenti sunt:
 min="10"
 max="100"

12.5.3.2. Procesarea documentelor XML care utilizează spații de nume

Pentru activarea facilității de a lua în considerare spațiile de nume, trebuie să invocăm metoda `setNamespaceAware()` cu valoarea `true` pentru obiectul de tip `DocumentBuilderFactory`. În acest caz se vor putea utiliza metodele următoare din interfața `Node`:

Prototipul metodei	Descrierea metodei
<code>String getLocalName()</code>	Returnează numele nodului curent neprefixat de indicatorul pentru spațiul de nume utilizat.
<code>String getNamespaceURI()</code>	Întoarce URI-ul spațiului de nume curent sau <code>null</code> , dacă acesta nu este specificat.
<code>String getPrefix()</code> throws <code>DOMException</code>	Returnează prefixul spațiului de nume pentru nodul curent sau <code>null</code> , dacă acesta nu este specificat.
<code>void setPrefix(String prefix)</code> throws <code>DOMException</code>	Modifică sau elimină prefixul nodului curent. În caz de eroare se aruncă o excepție <code>DOMException</code> .

Exemplul 12.5.3. Pentru a vedea cum sunt utilizate spațiile de nume în arborele DOM, vom considera următorul document XML, denumit `agenti.xml`:

```
<?xml version="1.0" ?>
<agenti>
  <agent xmlns="urn:infoiasi.ro:Agent" id="dil">
    <nume nr="9">Agent Web</nume>
    <proprietate>mobil</proprietate>
  </agent>
  <a:agent xmlns:a="urn:polirom.ro:Marketing" id="PR">
```

```
<a:nume a:nr="1">Agent</a:nume>
  <proprietate tip="generic">dinamic</proprietate>
</a:agent>
</agenti>
```

Pentru primul marcator `agent` s-a utilizat un spațiu de nume fără a utiliza un prefix, iar pentru cel de-al doilea s-a folosit prefixul `a`. Mai remarcăm faptul că al doilea tag `proprietate` nu face parte din nici un spațiu de nume, pe când primul este din spațiul de nume cu URI-ul `urn:infoiasi.ro:Agent`.

Programul următor va afișa la consolă, pentru fiecare nod de tip element sau atribut, prefixul și URI-ul spațiului de nume corespunzător, numele și valoarea nodului:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOMSpatiiNume {
  /** definirea unei metode private
   * pentru procesarea informatiilor din arbore */
  private static void procesare(Node nod) {
    /** afisare mesaj la consola */
    if (nod.getPrefix() != null || nod.getNamespaceURI() != null)
      System.out.println("NS: " + nod.getPrefix() +
        " - " + nod.getNamespaceURI());
    System.out.println(nod.getNodeName() +
      " - " + nod.getNodeValue());
    System.out.println();
    /** daca posedă atribute */
    if (nod.hasAttributes()) {
      NamedNodeMap atts = nod.getAttributes();
      for(int i=0; i<atts.getLength(); i++)
        procesare(atts.item(i));
    }
    /** daca are descendenți */
    if (nod.hasChildNodes()) {
      NodeList desc = nod.getChildNodes();
      Node n;
      for(int i=0; i<desc.getLength(); i++) {
        n = desc.item(i);
        /** ne interesează doar descendenții de tip element */
        if (n.getNodeType() == n.ELEMENT_NODE)
          procesare(n);
      }
    }
  }
}
```

```

public static void main(String[] args) {
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.
            newInstance();
        /** indicam faptul ca utilizam spatii de nume */
        factory.setNamespaceAware(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        /** crearea arborelui DOM */
        Document doc = builder.parse("spatii.xml");
        /** procesarea informatiilor din arbore */
        procesare(doc.getDocumentElement());
    } catch (Exception e) {
        System.err.println("Exceptie: " + e);
    }
}

```

Pentru fiecare nod de tip element sau atribut se afișează maximum două linii. Dacă este utilizat un spațiu de nume, atunci se afișează valoarea prefixului și a URI-ului spațiului de nume, iar ultima linie conține numele nodului și valoarea acestuia. Informațiile afișate corespunzătoare nodurilor sunt separate prin linii vide.

Execuția programului descris anterior va avea ca rezultat la consolă:

```

agenti - null

NS: null - urn:infoiasi.ro:Agent
agent - null

NS: null - http://www.w3.org/2000/xmlns/
xmlns - urn:infoiasi.ro:Agent

id - dil

NS: null - urn:infoiasi.ro:Agent
nume - null

nr - 9

NS: null - urn:infoiasi.ro:Agent
proprietate - null

NS: a - urn:polirom.ro:Marketing
a:agent - null

NS: xmlns - http://www.w3.org/2000/xmlns/
xmlns:a - urn:polirom.ro:Marketing

```

```

id - PR

NS: a - urn:polirom.ro:Marketing
a:nume - null

NS: a - urn:polirom.ro:Marketing
a:nr - 1

proprietate - null

tip - generic

```

Liniile care încep cu simbolurile NS specifică informațiile privitoare la spațiile de nume. Se observă faptul că primul spațiu de nume nu are prefix, al doilea marcator proprietate nu este inclus în nici un spațiu de nume ș.a.m.d.

12.5.3.3. Procesarea documentelor XML care conțin DTD

Pentru a indica faptul că documentul XML care se dorește a fi procesat trebuie să fie validat conform unui DTD specificat, se va apela metoda `setValidating()` cu valoarea parametrului `true` pentru obiectul `DocumentBuilderFactory`. Implicit, această facilitare nu este activată.

Este bine să stabilim un obiect a cărui clasă implementează interfața `Error Handler` pentru a trata eventualele erori apărute la validare. Astfel, putem avea controlul validării documentelor XML. Acest lucru se realizează prin apelul metodei `setErrorHandler()` din clasa `DocumentBuilder`, ca parametru transmițându-se obiectul care realizează tratarea erorilor.

Exemplul 12.5.4. Considerăm documentul XML din exemplul 12.5.3., în care inserăm definiția unui DTD corespunzător după prima linie:

```

<!DOCTYPE agenti [
    <!ELEMENT agenti (agent(a:agent)+>
    <!ELEMENT agent (nume,proprietate)>
    <!ATTLIST agent
        id CDATA #IMPLIED
        xmlns CDATA #IMPLIED
    >
    <!ELEMENT a:agent (a:nume,proprietate)>
    <!ATTLIST a:agent
        id CDATA #IMPLIED
        xmlns:a CDATA #IMPLIED
    >
    <!ELEMENT nume (#PCDATA)>
    <!ATTLIST nume

```

```

    nr    CDATA    #REQUIRED
  >
  <!ELEMENT a:nume (#PCDATA)>
  <!ATTLIST a:nume
    a:nr CDATA    #REQUIRED
  >
  <!ELEMENT proprietate (#PCDATA)>
  <!ATTLIST proprietate
    tipCDATA    #IMPLIED
  >
}>

```

Vom denumi fișierul astfel rezultat ca `agenti_dtd.xml`. Execuția programului Java de mai jos va afișa mesaje de eroare în cazul în care nu este respectată structura definită de DTD-ul specificat:

```

import org.w3c.dom.*;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import javax.xml.parsers.*;

public class DOMValidare {
    public static void main(String[] args) {
        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.
                newInstance();
            /** indicam faptul ca utilizam spatii de nume */
            factory.setNamespaceAware(true);
            /** cerem validarea documentului XML conform DTD-ului
                specificat */
            factory.setValidating(true);
            DocumentBuilder builder = factory.newDocumentBuilder();
            /** indicam obiectul pentru tratarea erorilor */
            MyErrorHandler err = new MyErrorHandler();
            builder.setErrorHandler(err);
            /** crearea arborelui DOM */
            Document doc = builder.parse("spatii_dtd.xml");
            /** daca nu au fost semnalate erori */
            if (!err.isEroare())
                System.out.println("OK");
        } catch (Exception e) {
            System.err.println("Exceptie: " + e);
        }
    }
}

```

```

    }
}

/** clasa definita pentru tratarea erorilor */
class MyErrorHandler implements ErrorHandler {
    /** declararea unei date private membre */
    boolean eroare = false;

    public void error(SAXParseException e) throws SAXException {
        System.err.println("Eroare: " + e.getMessage());
        eroare = true;
    }

    public void fatalError(SAXParseException e) throws
        SAXException {
        System.err.println("Eroare fatala: " + e.getMessage());
        System.exit(1);
    }

    public void warning(SAXParseException e) throws SAXException {
        System.err.println("Avertisment: " + e.getMessage());
        eroare = true;
    }

    /** Metoda pentru verificarea datei private "eroare" */
    public boolean isEroare() { return eroare; }
}

```

Data membră `eroare` a clasei `MyErrorHandler` va indica dacă a apărut sau nu vreo eroare la validarea documentului XML.

În cazul în care nu apar erori, programul va afișa mesajul OK la consolă, altfel va apărea un mesaj de eroare. De exemplu, dacă mai adăugăm un marcator agent care are conținut vid, atunci va fi afișat mesajul:

Eroare: Element "agent" requires additional elements.
care indică faptul că elementul agent trebuie să conțină sub-elemente.

12.5.3.4. Interfața Document

Interfața Document are o importanță deosebită, deoarece prin intermediul acesteia se pot crea noi noduri pentru arborele DOM asociat unui document XML. Aceasta mai oferă posibilitatea obținerii unei colecții de noduri care au același nume.

Cele mai semnificative metode ale interfeței Document sunt următoarele:

Prototipul metodei	Descrierea metodei
Element getDocumentElement()	Întoarce nodul rădăcină al arborelui.
Element getElementById (String id)	Returnează elementul care are identificatorul precizat sau null, dacă acesta nu există. Atributul id trebuie să fie definit ca fiind de tip ID.
NodeList getElementsByTagName (String tag)	Furnizează o colecție cu toate nodurile de tip element care au numele specificat.
NodeList getElementsByTagNameNS (String URI, String numeLocal)	Creează o colecție cu nodurile de tip element din spațiul de nume cu URI-ul dat și cu numele indicat.
Node importNode (Node nod, boolean adancime) throws DOMException	Importă nodul specificat din alt document în cel curent. Acesta nu va avea nici un nod părinte. Nodul sursă nu este afectat întrucât se creează o copie a acestuia. Dacă parametrul adancime are valoarea true, atunci se importă întreg subarborele dat de nodul importat, altfel se importă doar nodul specificat. În caz de eroare se aruncă o excepție DOMException.
Attr createAttribute (String nume) throws DOMException	Creează un nod de tip atribut cu nume specificat. Acesta se poate adăuga la un nod de tip element cu ajutorul metodei setAttributeNode() .
CDATASection createCDATASection (String data) throws DOMException	Creează un nod de tip secțiune CDATA cu conținutul specificat.
Comment createComment (String data)	Returnează un nou nod comentariu cu textul dat de parametrul data.
DocumentFragment createDocumentFragment()	Creează un nod vid de tip DocumentFragment.
Text createTextNode (String data)	Construiește un nod text.
Element createElement (String numeTag) throws DOMException	Întoarce un nou nod corespunzător marcatorului cu numele specificat.
EntityReference createEntityReference (String nume) throws DOMException	Întoarce un nou obiect de tipul EntityReference.
ProcessingInstruction createProcessingInstruction (String tinta, String data) throws DOMException	Creează un nod corespunzător unei instrucțiuni de procesare cu datele indicate.

Nodurile create de metodele interfeței Document nu sunt automat adăugate în arborele DOM corespunzător documentului XML. Acestea trebuie inserate în lista nodurilor descendente ale unui nod. Modificarea arborelui nu implică actualizarea fișierului sursă XML.

Exemplul 12.5.5. Fie următorul document XML:

```
<?xml version="1.0" ?>
```

```
<!DOCTYPE studenti [
  <!ELEMENT studenti (student)+>
  <!ELEMENT student (nume,an,grupa)>
  <!ELEMENT nume (#PCDATA)>
  <!ELEMENT an (#PCDATA)>
  <!ELEMENT grupa (#PCDATA)>
]>
<studenti>
<student>
  <nume>Popovici Paul</nume>
  <an>2</an>
  <grupa>3</grupa>
</student>
<student>
  <nume>Farcas Dumitru</nume>
  <an>2</an>
  <grupa>1</grupa>
</student>
<student>
  <nume>Cristea Marius</nume>
  <an>3</an>
  <grupa>1</grupa>
</student>
<student>
  <nume>Munteanu Daniel</nume>
  <an>2</an>
  <grupa>3</grupa>
</student>
<student>
  <nume>Cracana Roxana</nume>
  <an>3</an>
  <grupa>1</grupa>
</student>
</studenti>
```

Pentru adăugarea unui nou student, vom scrie următoarea secvență de cod:

```
/** adaugarea unui nou student */
Node stud = doc.createElement("student");
/** cream numele */
Node n = doc.createElement("nume");
Node t = doc.createTextNode("Mironescu Adrian");
n.appendChild(t);
stud.appendChild(n);
/** cream anul */
n = doc.createElement("an");
```

```

t = doc.createTextNode("3");
n.appendChild(t);
stud.appendChild(n);
/** cream grupa */
n = doc.createElement("grupa");
t = doc.createTextNode("2");
n.appendChild(t);
stud.appendChild(n);
/** adaugarea nodului student in arborele DOM */
doc.getDocumentElement().appendChild(stud);

```

Pentru a crea marcatorul nume cu textul Mironescu Adrian trebuie să creăm două noduri: unul corespunzător tagului, iar celălalt pentru nodul de tip text. Nodul tagului va avea ca descendent nodul cu text. Analog, se creează nodurile corespunzătoare marcatorilor an și grupa. Acești trei marcatori vor fi descendenți pentru nodul student.

Extragerea numelor tuturor studenților din arborele DOM se realizează astfel:

```

/** afisarea tuturor studentilor */
System.out.println("\t Lista tuturor studentilor:");
/** obtinerea listei de noduri corespunzatoare tagurilor nume */
NodeList lista = doc.getElementsByTagName("nume");
/** parcurgerea listei */
for(int i=0;i<lista.getLength();i++)
    /** afisarea continutului nodului descendent de tip text */
    System.out.println(lista.item(i).getFirstChild().getNodeValue());

```

Efectul execuției codului anterior va fi:

```

    Lista tuturor studentilor:
    Popovici Paul
    Farcas Dumitru
    Cristea Marius
    Munteanu Daniel
    Cracana Roxana
    Mironescu Adrian

```

Cu ajutorul arborelui DOM asociat putem modifica dinamic conținutul documentului XML. Se pot adăuga, modifica, respectiv șterge taguri, comentarii, texte etc.

12.6. XSLT

Transformările XSLT (eng. *XSL Transformation*) descriu un limbaj pentru transformarea documentelor XML în alte documente XML sau în alte documente de tip text. Acesta este definit de grupul de lucru pentru XSL al consorțiului Web. Recomandarea pentru XSLT 1.0 se găsește la adresa <http://www.w3c.org/TR/1999/REC-xslt-19991116>. Aici este disponibilă întreaga specificație pentru XSLT 1.0.

Transformarea se definește prin intermediul unui document XSLT, care este un tip particular de document XML. De regulă, acestea sunt păstrate în fișiere cu extensia .xsl.

În acest capitol vom considera documentul XML `optionale.xml` din secțiunea 12.4.2. ca document sursă ce urmează a fi transformat.

12.6.1. Pachete necesare

Pachetele necesare transformării documentelor XML sunt următoarele:

- `javax.xml.transform`
Conține clasele și interfețele necesare transformării documentelor XML.
- `org.xml.transform.dom`
Oferă clasele necesare specificării sursei de intrare sau ieșire prin DOM.
- `org.xml.transform.sax`
Oferă clasele necesare specificării sursei de intrare sau ieșire prin SAX.
- `org.xml.transform.stream`
Oferă clasele necesare specificării sursei de intrare sau ieșire prin intermediul fluxurilor de intrare/ieșire obișnuite.

12.6.2. Realizarea unei transformări

Unei transformări îi corespunde clasa `Transformer`. Înainte de crearea obiectelor din această clasă, va trebui să obținem un generator de transformări, adică o instanță la clasa `TransformerFactory`. Acest lucru se realizează prin apelul metodei statice `newInstance()` din aceeași clasă. Ambele clase sunt abstracte și fac parte din pachetul `javax.xml.transform`.

Obținerea unei instanțe la clasa `Transformer` se realizează prin apelul metodei `newTransformer()` din clasa `TransformerFactory`. Aceasta are două forme: fără parametru, caz în care se va realiza o copie a sursei de intrare, sau cu un parametru de tip `Source`, care va referi documentul cu definiția transformării.

Transformarea efectivă a unui document XML în alt tip de document este realizată de către metoda `transform()` din clasa `Transformer`. Prototipul metodei este următorul:

- `public abstract void transform(Source sursaXML, Result rezultat) throws TransformerException`

unde `sursaXML` desemnează documentul sau porțiunea de document XML care va fi transformată, iar `rezultat` indică unde este afișat rezultatul transformării.

`Source` și `Result` sunt două interfețe din pachetul `javax.xml.transform` utilizate pentru specificarea sursei de intrare, respectiv ieșirea transformării. Ambele posedă aceleași două metode:

Prototipul metodei	Descrierea metodei
<code>public String getSystemId()</code>	Returnează identificatorul stabilit cu metoda <code>setSystemId()</code> sau <code>null</code> , dacă aceasta nu s-a apelat.
<code>public void setSystemId (String id)</code>	Stabilește identificatorul pentru sursa de intrare, respectiv ieșire. Este utilizat pentru rezolvarea URI-urilor relative și în mesajele de eroare sau de avertizare.

Există trei clase care implementează interfața *Source*. Acestea sunt: *DOMSource*, *SAXSource* și *StreamSource*. De asemenea, există trei clase care implementează interfața *Result*, acestea fiind: *DOMResult*, *SAXResult* și *StreamResult*.

Sursele de intrare sunt date de interfața *Source*. Există trei tipuri de surse de intrare: obișnuite, surse DOM și surse SAX, iar pentru fiecare există câte o clasă predefinită care implementează interfața amintită.

Sursele obișnuite sunt specificate prin intermediul clasei *StreamSource*. Aceasta se află în pachetul `javax.xml.transform.stream`, iar cei mai semnificativi constructori ai acesteia sunt următorii:

Prototipul constructorului	Descrierea constructorului
<code>public StreamSource()</code>	Constructorul implicit creează un document vid.
<code>public StreamSource(File f)</code>	Se construiește o sursă de intrare dintr-un <code>File</code> .
<code>public StreamSource(InputStream flux)</code>	Se construiește o sursă de intrare dintr-un <code>InputStream</code> .
<code>public StreamSource(Reader r)</code>	Se construiește o sursă de intrare dintr-un <code>Reader</code> .
<code>public StreamSource(String url)</code>	Se construiește o sursă de intrare din URL-ul specificat.

Această clasă pune la dispoziție stabilirea sursei de intrare dintr-un `InputStream` sau `Reader`, respectiv obținerea acestora prin intermediul metodelor următoare:

- `public void setInputStream(InputStream flux)`
- `public void setReader(Reader r)`
- `public InputStream getInputStream()`
- `public Reader getReader()`

Exemplul 12.6.1. Programul Java va afișa pe ecran lista cursurilor opționale și a titularilor acestora. Informațiile sunt extrase din documentul `optionale.xml` definit în secțiunea 12.4.2, iar definiția transformării se află în fișierul `optionale_1.xsl`. Conținutul fișierului `optionale_1.xsl` este următorul:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:for-each select="//optional">
      <xsl:value-of select="nume" />,
      <xsl:value-of select="profesor" />.
    </xsl:for-each>
  </xsl:template>
  <xsl:output method="text" />
</xsl:stylesheet>
```

iar a fișierului `VizualizareCursuri.java` este:

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
```

```
public class VizualizareCursuri {
    public static void main(String[] args) {
        TransformerFactory factory = TransformerFactory.
            newInstance();
        Transformer trans;
        try {
            // crearea transformarii
            trans = factory.newTransformer(new StreamSource
                ("optionale_1.xsl"));
            // crearea sursei din fisierul optionale.xml
            StreamSource sursa = new StreamSource("optionale.xml");
            // rezultatul va fi afisat pe ecran
            StreamResult rezultat = new StreamResult(System.out);
            // se realizeaza transformarea
            trans.transform(sursa, rezultat);
        } catch (TransformerConfigurationException e) {
            System.out.println("Eroare la initializare!\n" + e);
        } catch (TransformerException e) {
            System.out.println("Eroare la transformare!\n" + e);
        }
    }
}
```

Transformarea definită în fișierul `optionale_1.xsl` specifică faptul că sunt selectate toate tagurile `optional`, iar pentru fiecare vor fi afișate informațiile din interiorul subtagurilor `nume` și `profesor`.

Programul anterior va afișa în urma execuției la consolă:

```
Programarea interfetelor utilizator,
  Sabin Corneliu Buraga.
Compilare paralela,
  Stefan Andrei.
Medii virtuale,
  Stefan Ciprian Tanasa.
```

Sursele DOM sunt reprezentate de clasa *DOMSource*. Aceasta permite ca datele de intrare să fie preluate dintr-un nod al documentului XML. Transformarea se va aplica întregului subarbore determinat de nodul specificat.

Clasa *DOMSource* face parte din pachetul `javax.xml.transform.dom` și posedă doi constructori importanți:

- `public DOMSource()`
- `public DOMSource(Node nod)`

Primul este implicit și are un parametru de tip *Node*, care indică nodul din arborele DOM asociat documentului XML care va fi transformat. Se oferă posibilitatea stabilirii ulterioare a nodului prin intermediul `setNode()`, precum și obținerea acestuia cu metoda `getNode()`.

Exemplul 12.6.2. Se va afișa pe ecran un document XML care va cuprinde doar primul opțional. Transformarea este definită în fișierul `copiere.xsl`, care are următorul conținut:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>
  <xsl:output indent="yes" />
</xsl:stylesheet>
```

Programul Java este următorul:

```
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

public class AfisareNod {
  public static void main(String[] args) {
    TransformerFactory tFactory =
      TransformerFactory.newInstance();
    DocumentBuilderFactory docFactory =
      DocumentBuilderFactory.newInstance();
    try {
      // obtinerea arborelui DOM
      DocumentBuilder builder =
        docFactory.newDocumentBuilder();
      Document document = builder.parse("optionale.xml");
      // obtinerea nodului corespunzator primului optional
      Node nod = document.getDocumentElement().
        getElementsByTagName("optional").item(0);
      // crearea transformarii
      Transformer trans = tFactory.newTransformer(new Stream-
        Source("copiere.xsl"));
      // crearea sursei dintr-un nod DOM
      DOMSource sursa = new DOMSource(nod);
      // rezultatul va fi afisat pe ecran
      StreamResult rezultat = new StreamResult(System.out);
      // se realizeaza transformarea
```

```
trans.transform(sursa, rezultat);
} catch (Exception e) {
  System.out.println("Eroare!\n" + e);
}
}
```

Transformarea dată poate fi utilizată ori de câte ori se dorește scrierea într-un flux de ieșire a unui document XML sau a unei porțiuni ale sale.

Execuția programului va afișa la consolă următorul document XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<optional id="C003">
  <nume>Programarea interfetelor utilizator</nume>
  <profesor>Sabin Corneliu Buraga</profesor>
  <an>3</an>
  <url>http://www.infoiasi.ro/~busaco/courses/piu/</url>
  <studenti min="15" max="60"/>
  <recomandari>
    <curs>Birotica</curs>
    <curs>Grafica</curs>
  </recomandari>
</optional>
```

12.7. Concluzii

Limbajul Java oferă un foarte bun suport pentru procesarea documentelor XML. Sunt disponibile ambele modalități de parsare: SAX și DOM. Fiecare prezintă avantaje și dezavantaje, utilizarea acestora fiind în funcție de natura documentelor XML și de cerințele pe care trebuie să le îndeplinească aplicația.

SAX este utilizat, în general, pentru documente de mari dimensiuni și care nu necesită accesul aleator la date. Parcurgerea este secvențială, de la început până la sfârșit, și nu presupune memorarea întregului fișier la un moment dat.

DOM este frecvent întâlnit în aplicațiile care necesită accesarea informațiilor din diferite porțiuni ale documentului XML. Memoria ocupată de arborele asociat nu va fi un dezavantaj foarte mare în acest caz. Din cauza consumului mare de memorie, se are în vedere ca documentele XML să nu fie de mari dimensiuni.

XSLT permite transformarea datelor XML în alte date XML sau text. De regulă, transformările sunt definite în documente separate, sunt tot XML și au extensia `.xsl`.

Pentru a defini un tip de document se pot utiliza DTD-urile. Acestea indică structura unui document XML. Pentru a realiza validarea datelor XML, la parsare va trebui activată această facilități. De asemenea, trebuie specificat dacă se dorește procesarea unui document care conține spații de nume.

12.8. Test grilă

Întrebarea 12.8.1. Care dintre afirmațiile următoare sunt adevărate?

- a) Documentele XML sunt binare.
- b) Procesarea documentelor XML se poate realiza doar utilizând limbajul Java.
- c) Structura unui document XML poate fi specificată într-un fișier extern.
- d) Utilizând SAX nu se pot procesa documente XML de mari dimensiuni.

Întrebarea 12.8.2. Clasa DefaultHandler implementează interfețele:

- a) ErrorHandler
- b) TagHandler
- c) ContentHandler
- d) SAXHandler

Întrebarea 12.8.3. Care sunt super-interfețele interfeței Text?

- a) CDATASection
- b) CharacterData
- c) Comment
- d) nu există această interfață

Întrebarea 12.8.4. Care sunt subinterfețele interfeței Node?

- a) NodeList
- b) ProcessingInstruction
- c) Document
- a) NamedNodeMap

Întrebarea 12.8.5. Care dintre afirmațiile referitoare la arborele DOM sunt false?

- a) Nu se pot adăuga noduri aparținând altui arbore DOM.
- b) Se pot elimina subarbori.
- c) Se pot înlocui nodurile existente cu altele noi.
- d) Nu pot exista două noduri adiacente de același tip și cu același nume.

12.9. Exerciții propuse

Exercițiul 12.9.1. Să se implementeze o clasă care oferă posibilitatea eliminării anumitor taguri și attribute dintr-un document XML. Pentru a testa funcționalitatea respectivei clase se va crea o aplicație în care utilizatorul va specifica prin intermediul unui formular tagurile și attributele care vor fi eliminate.

Exercițiul 12.9.2. Să se elaboreze un document XML cu informații privitoare la orarul săptămânal (se va construi și un DTD pentru validare). Se va realiza o aplicație pentru vizualizarea orarului și o unealtă pentru actualizarea acestuia.

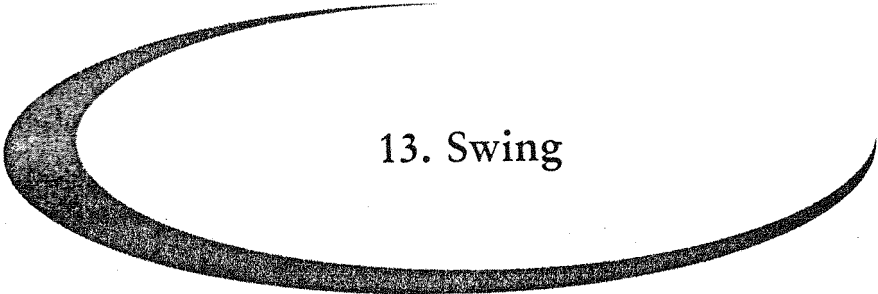
Exercițiul 12.9.3. Să se construiască un arbore DOM corespunzător arborelui genealogic. Vizualizarea grafică se va realiza de către un applet.

12.10. Proiecte propuse

Proiectul 12.10.1. Să se realizeze un program ce pentru orice document XML generează un DTD care să fie cel mai restrictiv. Eventual se va extinde aplicația pentru a genera un DTD pentru mai multe documente XML similare.

Proiectul 12.10.2. Să se construiască o aplicație care gestionează biblioteca personală. Datele vor fi stocate în fișiere XML și se vor implementa următoarele facilități:

- adăugarea/ștergerea unei cărți din evidență;
- modificarea informațiilor referitoare la o anumită carte;
- căutarea după unul sau mai multe criterii (autor, titlul cărții, editură, anul apariției);
- elaborarea unui raport referitor la cărțile împrumutate (se vor oferi și informații despre persoana căreia i s-a împrumutat cartea).



13. Swing

13.1. Cuvinte cheie

- interfețe grafice
- componente
- evenimente
- ascultători
- modele
- look-and-feel

13.2. Introducere

Librăria Swing este partea JFC (eng. *Java Foundation Classes*) care oferă componentele necesare programatorilor pentru crearea de interfețe grafice moderne, complexe și într-adevăr „prietenos” aplicațiilor Java. Swing aduce îmbunătățiri calitative față de mai vechiul pachet AWT (eng. *Abstract Window Toolkit*) folosit în trecut pentru realizarea interfețelor grafice, completându-i neajunsurile. Spre exemplu, în distribuția AWT pentru Java 1.0 programatorul avea la dispoziție numai patru fonturi cu care putea să opereze și un număr restrâns de tipuri de componente, insuficiente pentru crearea de interfețe cu un grad ridicat de complexitate. Totuși pachetul Swing nu este total separat de AWT cum s-ar putea crede, ci se poate spune că este construit peste acesta (eng. *on top*), așa cum se va observa din cuprinsul acestui capitol, AWT fiind folosit mai mult pe post de intermediar între Swing și sistemul de operare peste care este instalată platforma Java.

Printre îmbunătățirile aduse de Swing în afară de noile componente adăugate, putem enumera: mod de prezentare și comportare pentru interfață independentă de platformă, posibilitatea de a avea componente cu forme neregulare, accesibilitate pentru persoanele cu deficiențe etc. Acest pachet a fost astfel gândit încât pentru lucruri simple este nevoie de scrierea de cod puțin, iar pentru lucruri mai complexe, mărimea codului crește proporțional. De asemenea, se folosesc convenții logice pentru denumirea claselor, metodelor etc. care, o dată înțelese și învățate, determină creșterea vitezei de lucru.

13.2.1. Obiective urmărite în acest capitol

Datorită faptului că dezbateră un subiect precum Swing necesită spațiul unei cărți întregi, și nu al unui singur capitol, am încercat să surprindem în primul rând principiile fundamentale care stau la baza acestui pachet într-adevăr complex, a căror înțelegere dă posibilitatea folosirii pachetului Swing la adevărata lui valoare. De asemenea, vom dezbate fiecare componentă în parte, prezentându-i principalele caracteristici, cum ar fi localizarea componentei în cadrul pachetului Swing, modalități de construcție a componentelor, modalități de personalizare, evenimentele pe care componenta le poate genera, respectiv un exemplu de folosire efectivă a ei într-o aplicație având codul comentat, îndeajuns pentru ca cititorii să poartă la propriile lor implementări. Pentru descrierea completă a constructorilor, atributelor și metodelor fiecărei componente aveți la dispoziție *javadoc*-ul care reprezintă documentația standard pe care firma Sun o pune gratuit la dispoziția utilizatorilor. Sugerăm parcurgerea capitolelor într-o manieră încrucișată în funcție de trimiterile care apar în text, deoarece am folosit deseori „licențe”, prezentând în secvențele de cod componente care nu au fost descrise la acel moment, din dorința de a păstra codul cât mai complet.

13.2.2. Despre interfețe grafice

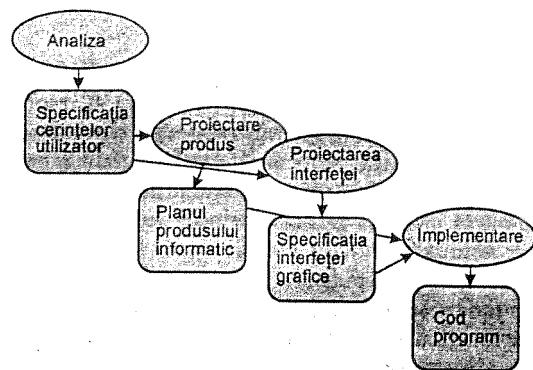
Interfața cu utilizatorul, pe scurt UI (eng. *User Interface*), reprezintă totalitatea mecanismelor care permit interacțiunea dintre o aplicație și utilizatorii ei. O particularizare a interfeței cu utilizatorul o reprezintă interfața grafică, pe scurt GUI (eng. *Graphic User Interface*), care se referă strict la comunicarea vizuală dintre utilizator și aplicație. Putem spune că interfața grafică este locul de întâlnire dintre utilizatori și aplicația care le este destinată.

Deseori aplicațiile sunt percepute de utilizatorii finali doar prin intermediul interfețelor acestora și de aceea o interfață prost realizată poate compromite întreaga aplicație chiar dacă funcțional ea este bine realizată. În acest sens, o problemă în procesul de realizare a interfeței grafice o reprezintă posibilitatea ca utilizatorul să nu înțeleagă cum să o utilizeze după ce a fost creată. Acest lucru se întâmplă mai ales pentru că, în general, utilizatorii unei aplicații (destinatarii aplicației) nu sunt și programatori având intenția de a crea interfața conform propriului model mental, diferit de cel al utilizatorilor. De aceea, dezvoltatorii aplicației trebuie să intuiască modalitățile prin care utilizatorii modelează conceptual diferitele funcționalități sau, în general, aspecte ale aplicației, și să încorporeze aceste modele în viitoarea interfață grafică. Metoda folosită în acest proces este analogia concretizată prin metaforă. Acolo unde analogiile sunt greu de realizat, sunt folosite idiomurile care reprezintă convenții între creatorii și utilizatorii interfeței, ușor de reținut de aceștia din urmă. Spre exemplu, funcția de ștergere într-o aplicație poate fi mai ușor concepută (modelată) de utilizatori ca aruncarea unui obiect nefolositor la coș. Astfel, pentru a șterge un icon, utilizatorul îl trage deasupra coșului de gunoi (eng. *trash*), reprezentat grafic de un alt icon, unde îl eliberează. Acest mod de a vedea lucrurile poartă numele de „înglobarea semanticii în sintaxă”, deoarece se încearcă o folosire a simbolurilor care sunt mai ușor de înțeles și reținut. Un alt exemplu ar putea fi folosirea de iconuri în barele de unelte, care ascund printr-o reprezentare grafică uneltele care se vor folosi după selectarea lor (spre exemplu, o lupă folosită mai apoi ca unelță pentru realizarea operației de micșorare-mărire). Se poate observa faptul că, în general, există un număr limitat de astfel de metafore și simboluri care se repetă în mai toate aplicațiile și pe toate platformele.

Firma Sun, cea care dă specificațiile limbajului Java, sugerează folosirea unui set standard de iconuri și idiomuri care sunt grupate într-o arhivă pe care o găsiți la adresa: developer.java.sun.com/developer/techDocs/hi/repository/.

Din punctul de vedere al ingineriei programării, momentul în care se pune cu adevărat problema interfeței cu utilizatorii este în procesul de proiectare arhitecturală a aplicației, dar creatorii aplicației vor face aprecieri chiar în procesul de analiză a cerințelor utilizatorilor. Așa cum se realizează un plan al produsului informatic, se recomandă realizarea planului interfeței cu utilizatorul prin care se clarifică cum va arăta interfața grafică înainte de a trece la implementarea efectivă a acesteia.

În procesul de proiectare a interfețelor grafice se poate folosi prototipizarea care presupune realizarea de prototipuri ale interfeței grafice (eventual, fără funcționalități care se pot adăuga mai apoi), prototipuri care sunt evaluate și reactualizate, dacă este cazul, pe întreg parcursul dezvoltării aplicației. Prototipurile interfeței grafice clarifică



interacțiunile aplicației cu utilizatorii sau cu alte aplicații, respectiv funcționalitățile aplicației. De asemenea, prin acest procedeu utilizatorii sunt atrași în procesul de realizare a aplicației, se pot descoperi unele lipsuri în cerințele utilizatorilor etc.

Alte informații despre principiile care stau la baza proiectării interfețelor grafice pentru aplicațiile Java puteți găsi la adresa: java.sun.com/products/jlrf/.

13.2.3. JFC (Java Foundation Classes)

JFC reprezintă un grup de API-uri (eng. *Application Programming Interface*) incluse în platforma J2SE (eng. *Java 2 Standard Edition*) care permit dezvoltatorilor să creeze interfețe grafice (GUI) aplicațiilor Java. Java Foundation Classes constă din cinci pachete: AWT, Swing, Accessibility, Java 2D și Drag and Drop, care se întrepătrund, neavând în general o existență de sine stătătoare. Java 2D a devenit o parte integrantă a AWT, Swing este construit peste AWT (se așteaptă ca Swing să se îmbine mult mai puternic cu AWT în următoarele distribuții Java), suportul Accessibility este construit în Swing, iar API-ul Drag and Drop este integrat în AWT și Swing. API-urile Java 2D și Drag and Drop sunt disponibile doar pentru platforma Java 2. Oferim în tabela care urmează o scurtă prezentare a celor cinci părți componente ale JFC:

Parte componentă a JFC	Explicație
AWT	Parte a JFC proiectată pentru a oferi suport pentru crearea interfețelor grafice folosind modul grafic. Oferă o ierarhie de clase suficiente pentru proiectarea unei interfețe grafice nesofisticate pentru un applet sau o aplicație. Reprezintă nucleul JFC și fundamentul pe care s-a construit pachetul Swing.
Swing	Reprezintă un set vast de componente grafice, de la cele foarte simple, cum ar fi etichetele sau butoanele, la cele foarte complexe ca, de exemplu, tabelele sau arborii. Oferă independență față de platforma peste care este instalată mașina virtuală și posibilitatea personalizării aspectului interfeței și a comportamentului acesteia la acțiunile utilizatorilor.

Parte componentă a JFC	Explicație
Accessibility	Reprezintă modalitatea de a oferi ajutor utilizatorilor cu deficiențe, cum ar fi posibilitatea de a descifra interfața folosind mesaje vocale, Braille sau dispozitive exterioare atașate calculatorului. Oferă informațiile de care au nevoie programatorii care scriu soft pentru astfel de dispozitive auxiliare sau look-and-feel-uri. Pachetul <code>javax.accessibility</code> este cel care stă la baza acestui sistem, dar cei care scriu programe obișnuite nu au nevoie să îl folosească.
Java 2D API	API-ul care permite utilizatorilor să încorporeze în aplicațiile lor grafică 2D complexă, text și imagini.
Drag and Drop	Oferă posibilitatea de a realiza operații de drag and drop între aplicația curentă și aplicațiile native, sau între două aplicații Java. La baza acestei facilități stau pachetele <code>java.awt.dnd</code> și <code>java.awt.datatransfer</code> , dar în general nu este nevoie de folosirea lor de către programatori, deoarece sistemul drag and drop funcționează implicit.

Diferența majoră dintre Swing și AWT este aceea că, spre deosebire de AWT, componentele Swing sunt scrise 100% în Java având ca bază API-ul JDK1.1, neconținând cod nativ (dependent de sistemul de operare). Aceasta înseamnă că butoanele Swing, spre exemplu, vor arăta și se vor comporta identic pe platforme Macintosh, Solaris, Linux sau Windows. Spre deosebire, butoanele AWT luau mereu aspectul platformei pe care rula aplicația. De asemenea, prin această independență față de platformă se obține o îmbunătățire a vitezei de execuție a aplicațiilor având interfețe realizate în Swing.

API-ul JFC a fost perfecționat și optimizat pe măsura apariției de noi distribuții ale platformei Java și îl regăsim ca parte integrantă în platforma Java 2 Standard Edition versiunile 1.2, 1.3, 1.4 și următoarele. Dacă folosiți vechiul JDK 1.1 va trebui să instalați și suportul pentru JFC, spre exemplu, distribuția JFC 1.1 pe care o găsiți la adresa java.sun.com/products/jfc. Unele clase și metode s-au învechit (eng. *deprecated*) și s-a renunțat la folosirea lor efectivă pe parcursul apariției noilor distribuții de platforme Java, din cauza ineficienței sau greșelilor pe care le conțineau. Pentru o listă detaliată a acestora, puteți consulta javadoc-ul la secțiunea *Deprecated*. Pentru a rula totuși aplicația conținând cod învechit în `j2sdk1.4`, codul trebuie recompilat folosind opțiunea `-deprecated`.

În particular, librăria Swing se poate folosi în aplicații dezvoltate pentru platforme care suportă JFC. Prin intermediul unui plug-in pentru browserele Web (purtând numele Java Plug-in) puteți să creați interfețe folosind Swing și pentru appleturi Java. Plug-in-ul, care a ajuns la versiunea 1.3, se poate descărca gratuit de la adresa java.sun.com/products/plugin. Pentru a vizualiza appleturile având interfețe realizate în Swing, fără să folosiți pentru aceasta un browser, aveți la dispoziție un utilitar numit `appletviewer` care vine o dată cu platforma Java și pe care îl găsiți în directorul `/bin` al distribuției.

13.2.4. Componentele și pachetele librăriei Swing

Componentele Swing sunt aproape toate derivate dintr-o singură clasă de bază numită `JComponent`, care moștenește la rândul ei clasa `Container` din AWT. Moștenirea clasei `JComponent` dă componentelor posibilitatea de a avea bordere, tooltips, respectiv look-and-feel configurabil. De asemenea, tot din `JComponent` se moștenesc și metodele prin care se pot seta dimensiunile și poziționarea componentelor, metode pe care le vom discuta în conținutul acestei secțiuni. În general, fiecare componentă grafică din AWT are o clasă echivalentă în Swing, al cărei nume se formează din echivalentul AWT la care adăugăm prefixul `J` (singura excepție este clasa `Canvas`). Reciproca nu este adevărată, deoarece Swing îmbogățește considerabil paleta componentelor disponibile în AWT.

O componentă grafică reprezintă un obiect care are o anumită reprezentare grafică ce poate fi afișată pe ecran și poate astfel interacționa cu utilizatorul. Reprezentări grafice au toate obiectele care sunt instanțe ale claselor derivate din `java.awt.Component` și, în particular, `javax.swing.JComponent` pentru componentele Swing. Pentru a crea un buton având un anumit text drept conținut, trebuie mai întâi să realizăm o instanță a clasei `JButton`:

```
JButton buton=new JButton("De acord");
```

Dacă dorim să creăm o listă de cuvinte din care utilizatorul să poată selecta un singur element la un moment dat, trebuie să realizăm mai întâi o instanță a clasei `JList` folosind unul dintre constructorii pe care această clasă îi pune la dispoziție.

```
String continut[]={"AWT", "Swing", "Accessibility", "Java 2D",
    "Drag and Drop"};
JList lista = new JList(continut);
```

În acest caz, butonul și lista reprezintă componente grafice, iar până aici modul de construcție a componentelor nu diferă prin nimic de construcția unui obiect obișnuit. Este de remarcat faptul că crearea obiectelor grafice nu determină și afișarea reprezentării lor grafice pe ecran. Componentele mai întâi trebuie să fie așezate pe o suprafață de lucru (care mai poartă denumirea de container), la anumite coordonate și ocupând o anumită zonă a acesteia. Acest lucru se face prin intermediul unui apel al containerului de forma `add(JComponent componenta)`. Trebuie înțeles faptul că nu se pot desena componente fără a fi adăugate inițial unei suprafețe de desenare reprezentate printr-un container de bază de tip `JFrame`, `Jdialog`, `Jwindow` sau `JApplet`.

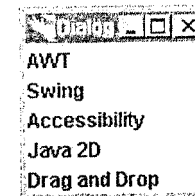
În continuare, vom adăuga o listă unei ferestre `JFrame`, care este una dintre componentele cu rol de container. Pentru mai multe informații despre suprafețele de desenare a se vedea capitolul „Container de bază”.

```
/*Fereastră*/
import javax.swing.*;
```

```
public class Fereastră extends JFrame {
    // constructorul clasei
    public Fereastră() {
        String continut[]={"AWT", "Swing", "Accessibility",
            "Java 2D", "Drag and Drop"};
        JList lista = new JList(continut); // am creat componenta
        this.getContentPane().add(lista); // o adaugam conținutului
        ferestrei
    }
}
```

După așezarea în container, componentele grafice vor deveni vizibile doar o dată cu vizualizarea întregii suprafețe de desenare printr-un apel `setVisible(true)`.

```
/*Aplicatia*/
import javax.swing.*;
public class Aplicatia {
    public static void main( String[] args ) {
        Fereastră fereastră = new Fereastră(); //cream fereastră
        fereastră.setVisible(true); // facem vizibila fereastră
    }
}
```



De asemenea, chiar dacă vizualizarea containerului a fost deja realizată, prin adăugarea unei noi componente containerului, aceasta va deveni la rândul ei vizibilă fără a mai fi nevoie de un apel `setVisible()`. Folosind un apel de forma `setVisible(false)` aplicat unei componente deja vizualizată, aceasta se va ascunde. Dacă este un container conținând și alte componente, acestea vor fi la rândul lor ascunse.

Modul în care componentele unui container vor fi desenate la momentul vizualizării acestuia depinde de gestionarul de poziționare care este asociat containerului. Acesta determină cum vor fi poziționate componentele pe suprafața de desenare și ce dimensiuni vor avea. Gestionarii sunt instanțe ale uneia din clasele `BorderLayout`, `BoxLayout`, `CardLayout`, `FlowLayout`, `GridLayout`, `GridBagLayout` sau noul gestionar `SpringLayout` introdus în distribuția 1.4. Toate aceste clase implementează interfața `LayoutManager` care le dă dreptul de a fi gestionari pentru containere. Începând cu versiunea JDK 1.1, a fost introdusă interfața `LayoutManager2` extinsă de cei mai mulți dintre gestionarii enumerați mai sus, care oferă acestora suport pentru aliniamentul componentelor conținute și gestiunea dimensiunilor maxime.

Prezentăm în continuare modalitatea prin care se atașează un gestionar unui container. Spre exemplu, deoarece gestionarul `BorderLayout`, care este asociat implicit containerului care ține conținutul unei ferestre, nu respectă dimensiunile asociate componentelor de către dezvoltator, îi vom asocia un alt gestionar de poziționare folosind metoda `setLayout()`.

```
JPanel continut=this.getContentPane();
continut.setLayout(new FlowLayout());
```

Dacă componenta nu este gestionată de un gestionar de poziționare (acest lucru se realizează printr-un apel `setLayoutManager(null)` pentru container), atunci ea va fi întotdeauna afișată cu dimensiunile și poziția care i-au fost atribuite în prealabil. Dacă nu i s-au asociat dimensiuni, implicit acestea vor fi 0, iar dacă nu a fost poziționată, va fi așezată în colțul din stânga sus al containerului.

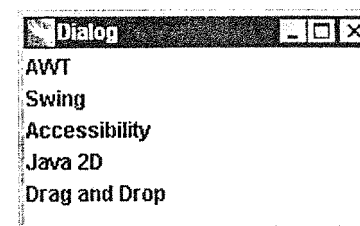
Este esențială deci înțelegerea modului în care lucrează gestionarii de poziționare, pentru a nu avea surprize la momentul desenării containerului. De asemenea, trebuie știut că implicit panourilor reprezentând containere intermediare li se asociază gestionarul `FlowLayout`, iar panoului conținut al unui container de bază i se asociază gestionarul `BorderLayout`. Se poate alege ca mai multe componente să fie grupate folosind containere intermediare (a se vedea capitolul „Containere intermediare”), astfel având posibilitatea de a le gestiona împreună poziționarea, separat de toate celelalte componente.

Fiecare componentă grafică are asociate trei dimensiuni (în fapt proprietăți) explicate în tabelul care urmează împreună cu metodele prin care acestea pot fi accesate:

Dimensiuni	Metode de acces	Explicații
PreferredSize	<code>getPreferredSize()</code> <code>setPreferredSize()</code>	Dimensiunea preferată a unei componente. Este folosită de cei mai mulți gestionari de poziționare pentru a dimensiona implicit componentele. Dimensiunea preferată a unei componente este determinată de look-and-feel-ul componentei și de fonturile pe care componenta le folosește.
MinimumSize	<code>getMinimumSize()</code> <code>setMinimumSize()</code>	Folosită de gestionarii de poziționare ca limită în micșorarea dimensiunilor componentelor.
MaximumSize	<code>getMaximumSize()</code> <code>setMaximumSize()</code>	Folosită de gestionarii de poziționare ca limită în maximizarea dimensiunilor componentelor.

Metodele de forma `setXXXSize()`, prezentate în capitolul precedent, primesc ca intrare o instanță a clasei `java.awt.Dimension`. În exemplul următor vom seta dimensiunea listei din exemplul precedent pentru a avea lățimea de 200 pixeli și înălțimea de 100 pixeli.

```
Dimension dim=new Dimension(200,100);
lista.setPreferredSize(dim);
```



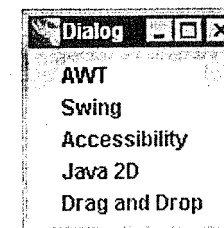
Pentru ca o componentă să fie desenată într-adevăr cu dimensiunile pe care le-am specificat în prealabil, trebuie să ne asigurăm că aceasta este gestionată de un gestionar de poziționare care respectă aceste setări. Spre exemplu, `FlowLayout` și `GridBagLayout` respectă dimensiunile preferate ale componentelor. Nu același lucru putem spune despre `BorderLayout` și `GridLayout`. Gestionarul `BoxLayout` este singurul gestionar care respectă dimensiunile maxime. Dacă se dorește redimensionarea unei componente care este deja vizibilă ea trebuie redesenată după setarea noilor dimensiuni. Pentru a aduce componentele la dimensiunile lor preferate (eng. *preferred size*), vom apela metoda `pack()` pentru container înainte de a apela `setVisible()`.

Pe lângă setarea dimensiunilor putem de asemenea modifica poziția fiecărei componente în interiorul containerului. Dacă componenta nu este gestionată de un manager de poziționare, acest lucru este ușor. În schimb, un manager de poziționare poate neglija dimensiunile pe care le dăm componentei. Pentru mai multe amănunte puteți consulta secțiunea dedicată gestionarilor de poziționare din capitolul dedicat appleturilor și componentelor AWT.

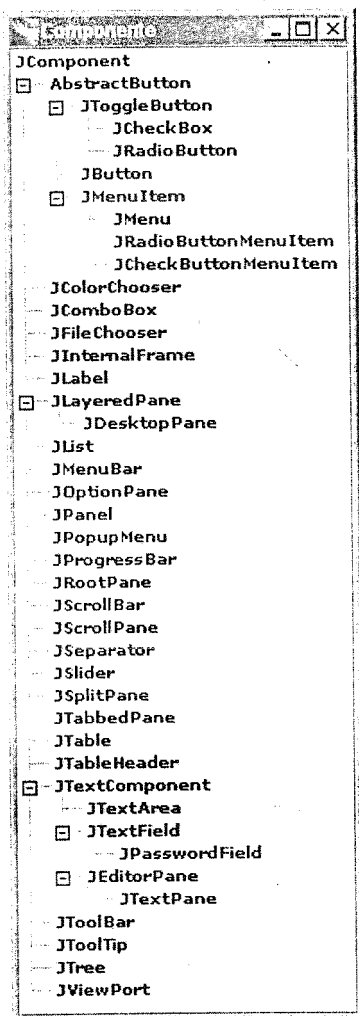
Metoda `setBounds()` moștenită din `JComponent` poate fi folosită pentru a seta în același timp mărimea și poziționarea unei componente în interiorul containerului părinte. Prezentăm în continuare cele două supraîncărcări ale metodei folosite în cazul exemplului anterior:

```
lista.setBounds(200,200,400,300);
```

```
Rectangle drept = new Rectangle(200,200,400,300);
lista.setBounds(drept);
```



Prezentăm în continuare lista parțială a componentelor Swing:



De asemenea, toate componentele grafice Swing sunt construite conform specificației JavaBeans, ceea ce înseamnă că sunt de fapt componente JavaBeans. Mai multe amănunte despre acest aspect găsiți în secțiunea „Conformanța cu JavaBeans”. Ce putem spune acum este că această compatibilitate dă posibilitatea construirii de medii vizuale pentru Java în care interfața grafică pentru aplicații se construiește interactiv prin așezarea componentelor grafice pe suprafața de lucru. Astfel, se pot testa efectiv mai multe posibilități de a proiecta interfața aplicației până se ajunge la o formă definitiv acceptată.

Pentru a putea folosi componentele Swing trebuie să importăm mai întâi pachetele corespunzătoare. În general este îndeajuns să includem următoarele pachete:

- javax.swing pentru componente Swing simple
- javax.swing.event pentru a asculta evenimente generate de componentele Swing
- java.awt pentru diverse aspecte legate de AWT
- java.awt.event pentru evenimente generale

Prezentăm lista completă a pachetelor Swing care trebuie incluse în aplicație în funcție de necesități.

Pachet	Prezentare
javax.swing	Conține componentele Swing de bază (prezentate în figura precedentă), modelele și interfețele implicite pentru componentele care dispun de ele. Pentru componentele complexe se definesc pachete separate.
javax.swing.border	Clase și interfețe folosite pentru a defini diferite stiluri de borduri (eng. <i>borders</i>). De remarcat faptul că bordurile nu sunt componente grafice.
javax.swing.colorchooser	Clase și interfețe care oferă suport pentru componenta JColorChooser care permite selectarea culorilor dintr-o paletă de culori.
javax.swing.event	Pachetul care conține evenimentele proprii Swing, respectiv ascultătorii (eng. <i>listeners</i>). Componentele Swing suportă, de asemenea, evenimente definite în pachetele java.awt.event și java.beans.
javax.swing.filechooser	Clasele și interfețele care oferă suport pentru componenta JFileChooser, cea cu ajutorul căreia se poate face selecția fișierelor.
javax.swing.plaf	Conține API-ul pentru look-and-feel prin care utilizatorii pot personaliza interfața componentelor. Cele mai multe clase din acest pachet sunt abstracte, implementatorii trebuind să le suprascrie pentru a crea diverse stiluri aplicabile interfețelor grafice.
javax.swing.plaf.basic	Conține clasele și interfețele pe baza cărora se construiesc pachete look-and-feel efective. Clasele din acest pachet trebuie suprascrise pentru a crea propriul pachet look-and-feel.
javax.swing.plaf.metal	Metal este look-and-feel-ul implicit al componentelor Swing.
javax.swing.plaf.multi	Acest pachet nu reprezintă un look-and-feel, ci oferă posibilitatea combinării mai multor look-and-feel-uri deja existente, putându-se astfel folosi simultan. Spre exemplu, astfel se poate combina un look-and-feel audio cu unul vizual.
javax.swing.table	Clase și interfețe care oferă suport pentru componenta JTable, care este folosită pentru reprezentarea datelor sub formă de tabele. Permite, de asemenea, personalizarea parțială a modului de vizualizare a tabelului fără ajutorul sistemului look-and-feel.
javax.swing.text	Clase și interfețe folosite de componentele text oferind suport pentru diverse aspecte ale acestora.
javax.swing.text.html	Suport pentru componentele text care vizualizează HTML.

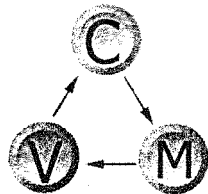
Pachet	Prezentare
javax.swing.text.html.parser	Suport pentru parsarea HTML.
javax.swing.text.rtf	Suport pentru documente RTF.
javax.swing.tree	Clase și interfețe care oferă suport pentru componenta JTree, care este folosită pentru reprezentarea datelor sub formă de arbore. Permite, de asemenea, personalizarea modului de vizualizare a arborelui fără ajutorul sistemului look-and-feel.
javax.swing.undo	Pachet care oferă suport pentru implementarea și gestionarea funcționalității undo/redo.

13.2.5. Principii de bază

Această secțiune descrie aspecte legate de modul în care a fost construită librăria Swing. Puteți începe direct cu următoarele secțiuni, care descriu componentele, și puteți reveni aici în cazul în care doriți să aprofundați. Pentru a înțelege cum funcționează componentele mai complexe, precum JTree sau JTable, este necesar studiul acestei secțiuni.

13.2.5.1. MVC (Model View Controller)

MVC este filosofia (nivelul conceptual) clasică pentru proiectarea interfețelor grafice în limbajele obiectuale care datează din anii 1970. Reprezintă structurarea componentelor în trei părți: un model, o vizualizare și un controler (eng. *controller*), fiecare având propria utilitate și funcționalitate și interacționând cu celelalte părți așa cum se poate vedea în figura care urmează:



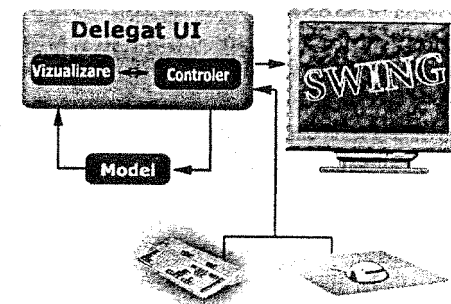
Rolurile părților unei componente sunt explicate în tabelul prezentat pe pagina următoare.

Componenta devine un mediator între model, vizualizare și controler. Unul dintre avantajele acestei modalități de a structura componentele este acela că ne permite să personalizăm look-and-feel-ul unei componente fără a modifica modelul. Putem să asociem unui același model (deci acelorași date) mai multe moduri de vizualizare chiar în același timp.

Partea componentă	Explicație
Model	Modelul păstrează în general datele cărora se dorește a li se da o vizualizare numindu-se, în acest caz, modele de date (eng. <i>data model</i>). Spre exemplu, pentru o componentă text, datele reprezintă textul ce se vrea vizualizat. Modelele comunică indirect cu vizualizarea și controlerul prin intermediul sistemului de evenimente și nu dețin referințe la nici una din acestea. Pot exista modele care să nu țină neapărat datele componente, ci stări ale vizualizării grafice pentru componentă (eng. <i>GUI-state</i>). Un exemplu în acest sens ar putea fi un model pentru o listă care încapsulează modul de selecție sau, pentru un buton, un model care încapsulează starea butonului: armat, apăsător etc. În cazul acestui ultim tip de modele, componentele dispun de metode pentru manipularea stării lor fără a face apel la modele. Trebuie precizat că deseori modelele încorporează atât datele componente, cât și stări ale vizualizării componentelor.
Vizualizare	Reprezentarea vizuală a modelului descriind pentru componentă modul cum arată (eng. <i>look</i>). Vizualizarea este responsabilă pentru a păstra pe ecran reprezentarea actualizată, primind în acest scop mesaje de la model și controler.
Controler	Controlerul ia datele de intrare ale utilizatorilor din vizualizare și le transformă în schimbări corespunzătoare în model. Controlerul este astfel responsabil pentru a determina când și cum va reacționa componenta la evenimentele de intrare dinspre dispozitivele de intrare cum ar fi mouse-ul și tastatura. În acest sens, reprezintă comportamentul (eng. <i>feel</i>) componente.

13.2.5.2. UI Delegate

Componentele Swing se bazează pe o versiune mai modernă a modelului de structurare MVC. Aceasta poartă denumirea de model-delegat și se deosebește de model-view-controller prin faptul că vizualizarea și controlerul componente sunt compactate într-o singură clasă care poartă denumirea de UI delegate. Această unire are loc deoarece este foarte greu să se scrie un controler generic care să nu știe nimic despre vizualizarea pe care o ascultă. Figura următoare explică acest model:



Pentru fiecare componentă Swing există câte o clasă UI delegate derivată din clasa abstractă `ComponentUI` prin care se gestionează look-and-feel-ul componente. Numele

acestei clase se formează de la numele componentei de la care se înlătură prefixul "J" și se adaugă sufixul "UI". Toate clasele delegat se află în pachetul `javax.swing.plaf`.

Metoda având prototipul `protected void setUI(ComponentUI newUI)`, definită în clasa `JComponent`, permite asocierea unui delegat unei componente.

Metoda având prototipul `static ComponentUI CreateUI(JComponent c)` este folosită pentru a obține o instanță a clasei delegat pentru componenta `c`.

Folosind aceste două metode putem, spre exemplu, să schimbăm look-and-feel-ul unei componente de tip `JButton` în felul următor:

```
JButton sterge=new JButton();
sterge.setUI((KunststoffButtonUI)KunststoffButtonUI.createUI(
sterge));
```

13.2.5.3. Look-and-feel

Swing permite dezvoltatorilor crearea de pachete care să conțină clasele delegat pentru mai multe componente sau toate tipurile componentelor pe care Swing le pune la dispoziție. Aceste pachete se numesc implementări pluggable look-and-feel (PLAF) și cu ajutorul lor se poate schimba stilul interfeței grafice chiar în momentul în care aceasta rulează (eng. *run-time*).

Pachetul `javax.swing.plaf` conține clasele abstracte (corespunzătoare componentelor clasice Swing), care implementează clasa `ComponentUI`, acestea fiind la rândul lor extinse în pachetul `javax.swing.plaf.basic`. Acest ultim pachet nu este un look-and-feel utilizabil, ci stă la baza implementărilor efective ale dezvoltatorilor de look-and-feel.

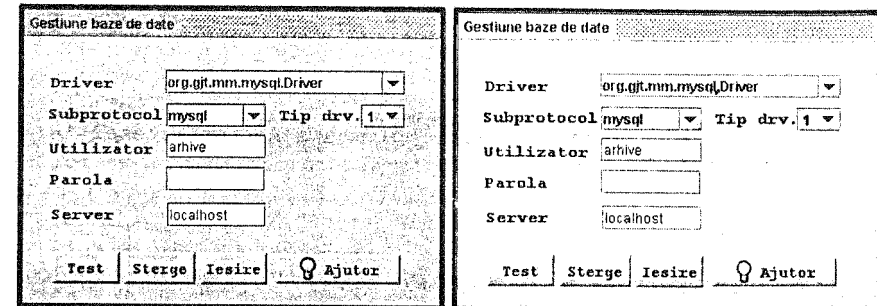
Există trei implementări care au la bază Basic look-and-feel, care vin o dată cu distribuțiile de platforme Java și se află în pachetele:

Windows: `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
 CDE\Motif: `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
 Metal (default): `javax.swing.plaf.metal.MetalLookAndFeel`

Java 2 pune la dispoziția posesorilor de sisteme Macintosh pachetul care simulează interfața utilizator Macintosh, dar acesta trebuie descărcat separat de pe rețea. Metal este look-and-feel-ul implicit al interfețelor, iar Windows și Macintosh sunt suportate doar pe platformele corespunzătoare.

De asemenea, există deja o serie de implementări realizate de terțe părți. O listă a lor o poți găsi la adresa: www.javatoo.com, de unde le puteți și descărca. Sunteți încurajați în scrierea de plug-in-uri pentru interfețele voastre grafice, având în acest sens la dispoziție un ghid prețios Java Look-and-Feel Design Guidelines despre care puteți afla mai multe la adresa: java.sun.com/products/jlff/at/book/index.html.

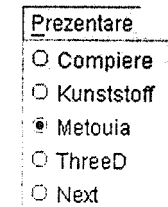
Prezentăm două vizualizări ale unei aceleiași ferestre de dialog, dar folosind stiluri diferite. Primul, Metal look-and-feel, iar cel de al doilea, Kunststoff, pe care îl puteți descărca gratuit de la adresa prezentată mai sus. A se vedea nuanțele de gri din a doua fereastră.



Pentru a putea schimba stilul aplicației, chiar și în timpul rulării, trebuie să căutați în pachetul ce conține plugg-in-ul o clasă derivată din `javax.swing.LookAndFeel`, al cărei nume este format din rădăcina `LookAndFeel` la care se adaugă prefixul ce dă numele plug-in-ului (de exemplu, `MetalLookAndFeel`). Această clasă face legătura între plug-in și aplicație. De asemenea, setarea plug-in-ului curent se face prin intermediul clasei `javax.swing.UIManager`, după cum putem vedea în exemplul următor prin intermediul căruia se setează ca plug-in curent *Motif*:

```
try {
    UIManager.setLookAndFeel(
        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    // am setat look-and-feel-ul Motif interfeței grafice
    SwingUtilities.updateComponentTreeUI(this);
    // această metodă este folosită pentru a imprima același
    // look-and-feel și
    // componentelor continute în containerul curent
}
catch (Exception e) {
    System.err.println("Nu se poate utiliza acest LookAndFeel");
}
```

Prezentăm în continuare o clasă pentru gestionarea plug-in-urilor prin încărcarea dinamică a claselor de apelare:



```
/* Clasa care genereaza un meniu permitand alegerea
look-and-feel-ului aplicatiei */
import javax.swing.*;
```

```

import java.awt.event.*;
import java.io.*;
// asigurați-va ca ati adaugat in CLASSPATH calea spre aceste
// pachete
public class LF extends JMenu implements ActionListener{
    JFrame f; // referinta catre fereastra careia i se va aplica
    plug-in-ul
    String[] lf = { "Compiere", "Kunststoff", "Metouia", "ThreeD",
    "Next"};
    String[] pachete = {"org.compiere.plaf.", com.incors.plaf.
    kunststoff.", net.sourceforge.mlf.metouia.", "swing.addon.
    plaf.threeD.", "nextlf.plaf."};
    public LF(String name, JFrame ff) {
        super(name);
        f=ff;
        ButtonGroup group = new ButtonGroup(); // clasa care
        permite selectarea unui singur buton
        for (int i = 0; i < lf.length; i++) {
            JRadioButtonMenuItem optiune = new JRadioButtonMenu
            Item(lf[i]);
            group.add(optiune); // adaugam optiunea curenta
            grupului de butoane
            add(optiune); // adaugam optiunea curenta meniului
            optiune.setActionCommand(i+"");
            optiune.addActionListener(this); // adaugam fiecarei
            optiuni ascultatorul
        }
    }

    public void actionPerformed(ActionEvent e) {
        try {
            int poz=Integer.parseInt(e.getActionCommand());
            String nume=pachete[poz]+lf[poz]+"LookAndFeel"; // nume
            poarta nume complet L&F
            Class clasa = Class.forName(nume); // folosind Reflection
            // API cream o clasa L&F
            LookAndFeel l=(LookAndFeel)clasa.newInstance(); // cream o
            // instanta a clasei
            UIManager.setLookAndFeel(l); // setam look-and-feel-ul
            // aplicatiei
            SwingUtilities.updateComponentTreeUI(f);
        } catch (Exception ex) {
            System.out.println("Eroare la incarcare plug-in"+
            ex.toString());
        }
    }
}

```

Folosind clasa descrisă mai sus puteți da utilizatorului aplicației posibilitatea să-și schimbe stilul interfeței grafice prin intermediul unui meniu cu butoane radio, chiar în timpul rulării programului:

```

final JMenu meniuTeme = new LF("Prezentare", this);
optiuni.add(meniuTeme);

```

13.2.5.4. Mai multe despre modele

Un alt avantaj pe care sistemul MVC îl oferă este posibilitatea de a personaliza modelul de date, acest lucru fiind posibil datorită despărțirii lui de vizualizare. Spre exemplu, putem crea zone de text (eng. *text area*) care să accepte numai introducerea de cifre, un anumit număr de caractere sau să nu reacționeze la apăsarea unor taste cum ar fi *Enter* sau *Tab* (a se vedea secțiunea dedicată componentelor text).

Prezentăm lista interfețelor care reprezintă modele și corespunzător componentele de care sunt utilizate, respectiv structurile de date pe care le înfășoară:

Interfața model	Componentele care folosesc acest model	Structura de date pe care o înfășoară
BoundedRange Model	JProgressBar, JScrollbar, JSlider.	4 întregi: value, extent, min, max cu următoarele restricții: value și extent trebuie să fie inițializate înainte de a specifica valorile min și max; extent este întotdeauna $\leq \max$ și $\geq \text{value}$.
ButtonModel	Toate subclasele componentei AbstractButton.	O valoare booleană reprezentând starea în care se află butonul (dacă butonul este selectat sau nu).
ListModel	JList.	O colecție de obiecte.
ComboBoxModel	JComboBox.	O colecție de obiecte și un obiect selectat.
MutableComboBox Model	JComboBox.	Un Vector sau o altă colecție de obiecte și un obiect selectat.
ListSelection Model	JList, Table ColumnModel.	Unul sau mai mulți indici corespunzând elementelor selectate dintr-un tablou sau listă. Permite selectarea unui singur element, interval sau a mai multor intervale.
SingleSelection Model	JMenuBar, JPopupMenu, JMenuItem, JTabbedPane.	Indexul elementului selectat dintr-o colecție gestionată de cel care implementează.
ColorSelection Model	JColorChooser.	O instanță a clasei Color reprezentând o culoare.
TableModel	JTable.	O matrice bidimensională de obiecte.
TableColumn Model	JTable.	O colecție de obiecte TableColumn, un set de ascultători pentru evenimentele legate de coloane, distanța dintre coloane, lățimea coloanelor, un model pentru selecție și un indicator pentru coloane selectate.

Interfața model	Componentele care folosesc acest model	Structura de date pe care o înfășoară
TreeModel	JTree.	Structuri arborescente. Implementările trebuie să fie capabile să distingă nodurile frunză de cele interioare.
TreeSelectionModel	JTree.	Nodurile selectate. Sunt permise selecții continue și discontinue.
Document	Toate componentele text.	Conținut sub formă de text. Implementările mai complexe suportă documente formate.

Componentele Swing care au modele permit obținerea unei referințe spre acestea sau adăugarea unui model nou prin intermediul metodelor `getModel()`, respectiv `setModel()`. Dacă nu se setează un model în momentul construcției componentei, un model implicit este creat și instalat intern în componentă. Prin convenție, acesta are un nume format din prefixul "Default" adăugat numelui modelului. Pentru componentele mai complexe, sunt puse la dispoziția programatorilor modele abstracte pentru ca aceștia să nu fie nevoiți să creeze modele pornind de la nimic. Numele acestora se formează din prefixul "Abstract" adăugat numelui modelului. A se vedea secțiunea dedicată componentei `JTable` pentru exemplificare.

Modelele nu dețin informații despre vizualizările componente în spatele cărora stau. Această abordare este necesară pentru a permite mai multe vizualizări pentru un același model. Un model nu deține decât o mulțime de ascultători atașați și interesați în a ști când se modifică starea modelului. Componenta este cea care prin intermediul ascultătorilor modelului își modifică vizualizarea redesenându-se.

Evenimentele aruncate de modele în momentul schimbării stării pot fi de două feluri. Pot fi evenimente „ușoare”, care nu poartă cu ele informații despre modificările survenite în model, ci se așteaptă ca ascultătorii să trimită un mesaj modelului și să afle ce s-a modificat. Aceste tipuri de evenimente sunt folosite în cazul în care starea componente se modifică des și sunt prezentate în tabelul următor:

Model	Ascultător	Eveniment
BoundedRangeModel	ChangeListener	ChangeEvent
ButtonModel	ChangeListener	ChangeEvent
SingleSelectionModel	ChangeListener	ChangeEvent

Singura informație pe care un astfel de eveniment o conține este sursa care l-a generat. Prezentăm un exemplu de tratare a unui astfel de eveniment:

```

JSlider slider = new JSlider();
BoundedRangeModel model = slider.getModel();
model.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent e) {
        BoundedRangeModel m = (BoundedRangeModel)e.getSource();
        // am obtinut o referinta la generatorul evenimentului
        System.out.println("Modelul este " + m.toString());
    }
});

```

Alte tipuri de evenimente sunt cele care descriu precis ceea ce s-a întâmplat în model atunci când au fost generate. O listă a acestora o prezentăm în continuare:

Model	Ascultător	Eveniment
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModelListener	TableColumnModelEvent
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

Prezentăm un exemplu de folosire a unui astfel de ascultător în cazul modelului `ListSelectionModel`, în care afișăm indexul elementului selectat, obținut din evenimentul generat:

```

String elemente[] = {"Unu", "Doi", "Trei"};
JList lista = new JList(elemente);
ListSelectionModel model = lista.getSelectionModel();
model.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        // obținem informații de la evenimentul aruncat
        if (!e.getValueIsAdjusting()) {
            System.out.println("S-a schimbat selecția: "
                + e.getFirstIndex());
        }
    }
});

```

Nu toate componentele au modele. În general, cele care au funcționalitate de container, cum ar fi `JFrame`, `JDesktopPane`, nu dispun de modele. La fel, componentele simple sau complexe în spatele cărora nu sunt multe informații. Au în schimb cele interactive, cum ar fi `JButton`, `JTextField`, `JTable`. De asemenea unele componente au mai multe modele, fiecare din ele participând la o parte din modelarea componente. Spre exemplu, `JTable` are un model pentru a păstra informații referitoare la selectare și un model pentru a păstra datele. Un același model poate fi folosit de mai multe componente așa cum se întâmplă în cazul modelului `BoundedRangeModel` folosit atât de `JSlider`, cât și de `JScrollbar`. După cum se observă, separarea modelului nu este o regulă atât de des întâlnită în Swing cum este separarea vizualizării și comportamentului.

13.2.5.5. Swing versus AWT

Componentele din AWT sunt numite componente „grele” (eng. *heavyweights*) în timp ce cele din Swing (în afară de containerele derivate direct din clasele AWT corespondente) sunt numite componente „ușoare” (eng. *lightweights*). Desenarea componentelor grele se face prin intermediul unor componente peer (eng. *asociate*) ale sistemului de operare, fiind astfel dependentă de sistemul peste care este instalată platforma Java. Aceste componente peer dau și look-and-feel-ul componentelor corespunzătoare din AWT și de aceea interfețele realizate în AWT au întotdeauna look-and-feel-ul sistemului de operare. În Swing, prin independența față de aceste componente native peer, componentele își gestionează singure look-and-feel-ul (sau mai bine spus dau utilizatorului această posibilitate), făcând astfel ca modul cum arată și cum se comportă o interfață grafică să fie identic, indiferent de sistemul de operare sau mașina pe care rulează aplicația. Și, în plus, dau posibilitatea de a schimba look-and-feel-ul în mod dinamic chiar în timpul rulării aplicației.

Din cauza acestor diferențe dintre aplicații, atunci când adăugăm o componentă grea într-un container, aceasta va acoperi toate componentele ușoare deja existente acolo. De asemenea, componentele grele nu permit transparența. De aceea trebuie să avem grijă atunci când utilizăm împreună într-o aceeași aplicație componente AWT și Swing. Clasa `java.awt.Toolkit` face legătura dintre componentele „grele” cum ar fi cele AWT și componentele corespondente peer specifice sistemului de operare pe care este instalată platforma Java. Această clasă reprezintă fundamentul pe care este construit pachetul AWT. În aplicații vom folosi această clasă doar pentru a obține rezoluția cu ajutorul căreia vom realiza poziționarea absolută a containerelor de bază.

13.3. Fundamentele Swing

13.3.1. Evenimente și ascultători

În Swing, interfața grafică (reprezentată de componentele grafice componente) este separată clar de implementare, care reprezintă codul ce se dorește a fi executat în urma interacțiunii utilizatorului cu interfața. Din acest motiv, prin construcție, Swing a fost conceput ca un sistem condus de evenimente. Evenimentele sunt generate (eng. *fire*) de fiecare dată când apăsăm o tastă sau un buton al mouse-ului și sunt tratate corespunzător prin cod implementat de programator. Acest mod în care componentele aruncă evenimente și felul în care acestea sunt tratate a rămas neschimbat de la platforma JDK 1.1 în care a fost implementat pentru prima oară.

Sunt multe tipuri de evenimente pe care componentele Swing le pot genera. Fiecare eveniment este înfășurat într-un obiect derivat din clasa `java.util.EventObject` prin intermediul căruia programatorul mănuieste evenimentul, putând determina sursa evenimentului (componenta care îl generează) și, uneori, informații despre

tipul evenimentului, starea sursei înainte și după ce evenimentul a fost generat. Sursele evenimentelor sunt în general componente sau modele, dar există și alte tipuri de obiecte care generează evenimente. În cazul evenimentelor derivate din clasa `EventObject`, sursa se determină efectiv folosind metoda public `Object getSource()`. Informații despre tipul evenimentului se obțin printr-un apel al metodei public `String toString()`. Evenimentele sunt grupate în două pachete: clasicul `java.awt.event`, care se referă la evenimente generale, respectiv `java.swing.event`, care se referă la evenimentele specifice componentelor grafice Swing.

Pentru a trata evenimentul generat de o componentă, programatorul trebuie să asocieze acesteia un obiect ascultător (eng. *listener*) care conține descrierea reacției aplicației la apariția evenimentului respectiv. Dacă evenimentul `xx` are tipul `XXEvent`, atunci ascultătorul este o implementare a interfeței `XXListener`, care se găsește de asemenea în unul din pachetele `java.awt.event` sau `java.swing.event`.

Prezentăm lista evenimentelor de bază din Swing, a ascultătorilor, a metodelor de adăugare, ștergere, respectiv componentele pentru care se aplică. Desigur există și alte evenimente derivate din cele de bază, specifice anumitor componente.

Evenimente, ascultători și metodele de adăugare-ștergere	Componentele care suportă aceste evenimente
ActionEvent ActionListener <code>addActionListener()</code> <code>removeActionListener()</code>	JButton, JList, JTextField, JMenuItem și componentele derivate inclusiv JCheckBoxMenuItem, JMenu și PopupMenu .
AdjustmentEvent AdjustmentListener <code>addAdjustmentListener()</code> <code>removeAdjustmentListener()</code>	JScrollbar și orice implementare a interfeței Adjustable .
ComponentEvent ComponentListener <code>addComponentListener()</code> <code>removeComponentListener()</code>	Component și derivatele incluzând JButton, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea și JTextField .
ContainerEvent ContainerListener <code>addContainerListener()</code> <code>removeContainerListener()</code>	Container și derivatele incluzând JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog și JFrame .
FocusEvent FocusListener <code>addFocusListener()</code> <code>removeFocusListener()</code>	Component și derivate .
KeyEvent KeyListener <code>addKeyListener()</code> <code>removeKeyListener()</code>	Component și derivate .

Evenimente, ascultători și metodele de adăugare-ștergere	Componentele care suportă aceste evenimente
MouseEvent (folosit și pentru click și pentru mișcare) MouseListener addMouseListener() removeMouseListener()	Component și derivate.
MouseEvent (folosit și pentru click și pentru mișcare) MouseMotionListener addMouseMotionListener() removeMouseMotionListener()	Component și derivate.
WindowEvent WindowListener addWindowListener() removeWindowListener()	Window și derivatele incluzând JDialog, JFileDialog și JFrame.
ItemEvent ItemListener addItemListener() removeItemListener()	JCheckBox, JCheckBoxMenuItem, JComboBox, JList și toate implementările interfeței ItemSelectable.
TextEvent TextListener addTextListener() removeTextListener()	Orice derivare din JTextComponent, incluzând JTextArea și JTextField.

Tratarea evenimentului se face prin suprascrierea metodelor (cel puțin una) care sunt descrise în interfețe și care primesc ca unic parametru evenimentul de tipul `XXEvent` generat de componenta ascultată de ascultător. Prezentăm o listă a evenimentelor și a metodelor corespunzătoare ce trebuie suprascrise:

Interfața ascultător/ adaptor	Metodele interfeței
ActionListener	actionPerformed (ActionEvent)
AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden (ComponentEvent) componentShown (ComponentEvent) componentMoved (ComponentEvent) componentResized (ComponentEvent)
ContainerListener ContainerAdapter	componentAdded (ContainerEvent) componentRemoved (ContainerEvent)
FocusListener FocusAdapter	focusGained (FocusEvent) focusLost (FocusEvent)
KeyListener KeyAdapter	keyPressed (KeyEvent) keyReleased (KeyEvent) keyTyped (KeyEvent)

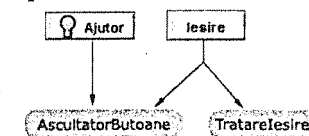
Interfața ascultător/ adaptor	Metodele interfeței
MouseListener MouseAdapter	mouseClicked (MouseEvent) mouseEntered (MouseEvent) mouseExited (MouseEvent) mousePressed (MouseEvent) mouseReleased (MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged (MouseEvent) mouseMoved (MouseEvent)
WindowListener WindowAdapter	windowOpened (WindowEvent) windowClosing (WindowEvent) windowClosed (WindowEvent) windowActivated (WindowEvent) windowDeactivated (WindowEvent) windowIconified (WindowEvent) windowDeiconified (WindowEvent)
ItemListener	itemStateChanged (ItemEvent)

După cum se poate vedea și din tabel, ascultătorul poate fi și o extensie a unei clase adaptor, având numele `XXAdaptor`, care este o implementare abstractă a interfeței `XXListener` pe care limbajul Java o pune la dispoziție cu scopul ca programatorul să poată evita suprascrierea tuturor metodelor interfeței. În general adaptorii există pentru ascultătorii care au mai mult de o metodă ce trebuie implementată. Clasele adaptor le găsiți în aceleași pachete unde se află și interfețele pe care acestea le extind. Pentru a adăuga sau șterge ascultătorii unei componente se folosesc metodele `addXXListener(XXListener ascultator)`, respectiv `removeXXListener(XXListener ascultator)`. Adăugarea de ascultători, respectiv ștergerea lor se poate face chiar și dinamic, în timpul rulării aplicației.

În general componentele permit înregistrarea mai multor ascultători pentru un același eveniment (eng. *multicast mode*). Dacă componenta nu permite acest lucru (eng. *unicast mode*) și forțăm atașarea mai multor ascultători, se va arunca o excepție `TooManyListenersException`. Nu se garantează execuția ascultătorilor în ordinea în care au fost adăugați. De asemenea, un același ascultător poate fi asociat oricărui componente care sunt surse de evenimente pe care el le poate capta. Uneori clasele care suportă evenimente de tipul `XXEvent` pun la dispoziția utilizatorului metode de genul `fireXX()` prin care programatorii pot genera singuri evenimente de tipul `XXEvent`, ce sunt trimise mai apoi spre tratare ascultătorilor asociați.

Este foarte important ca metodele ascultătorilor să se execute rapid, deoarece mănuierea evenimentelor și desenarea componentelor sunt executate în același fir de execuție care poartă numele de event despatching thread.

Prezentăm ca prim exemplu două butoane și tratarea operației de apăsare a lor:



```

import java.awt.event.*;
// clasa care contine evenimentele si interfetele listenerilor
//...
JButton iesire = new JButton();
JButton ajutor = new JButton();
//...
// definirea clasei ascultator care trateaza evenimentul generat
// de apasarea butonului
public class AscultatorButoane implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String numeButon=e.getActionCommand();
        System.out.println("Ati apasat butonul "+numeButon);
    }
}
// adaugam ascultatorul comun celor doua butoane
iesire.addActionListener(new AscultatorButoane());
ajutor.addActionListener(new AscultatorButoane());
//definirea ascultatorului tratareIesire
public class TratareIesire implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        dispose(); // inchide fereastra curenta
    }
}
// atasarea unui nou ascultator butonului Iesire
iesire.addActionListener(new TratareIesire());

```

Putem implementa ascultătorii folosind clase anonime, aceasta fiind metoda cea mai recomandată, deoarece se compactează la un loc și declararea ascultătorului, și atașarea lui componente. Prezentăm o rescriere a ascultătorului `AscultatorButoane` din exemplul precedent:

```

iesire.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String numeButon=e.getActionCommand();
        System.out.println("Ati apasat butonul "+numeButon);
    }
});

```

S-ar putea crede că în cazul ascultătorilor anonimi pierdem posibilitatea de a adăuga un același ascultător mai multor componente, dar această problemă se poate rezolva prin încapsularea funcționalității ascultătorului într-o metodă aparte. Acest artificiu ne dă posibilitatea de a ascunde prin folosirea unui modificador `private` tratarea evenimentului, deoarece metodele din interfețele ascultătorilor care trebuie suprascrise sunt în general publice. A se vedea exemplul care urmează:

```

private void functionalitate(ActionEvent e)
{
    String numeButon=e.getActionCommand();
}

```

```

        System.out.println("Ati apasat butonul "+numeButon);
    }
}
iesire.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    { functionalitate(e);
    }
});
ajutor.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    { functionalitate(e);
    }
});

```

Această metodă poate fi folosită pentru a adăuga un același ascultător mai multor componente de tipuri diferite. Spre exemplu, aceeași operație de copiere (eng. *copy*) poate fi făcută și dintr-un meniu derulant (eng. *pop-up*), ca și dintr-o opțiune a meniului bară, suplinind astfel folosirea clasei `AbstractAction`.

O componentă poate fi făcută ea însăși ascultător pentru evenimentele pe care le generează sau componentele conținute. Pentru aceasta trebuie să implementeze interfața corespunzătoare tipului de eveniment ascultat, să dea implementare metodei care tratează evenimentul și să-l asocieze ca ascultător. Prezentăm exemplul unei ferestre care-și gestionează singură operația de închidere:

```

public class FereastrAplicatie extends JFrame implements
ActionListener, WindowListener{
    public FereastrAplicatie() throws Exception {
        super("FereastrA");
        JButton iesire = new JButton();
        getContentPane.add(iesire);
        iesire.addActionListener(this);
        this.addWindowListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println("Ati iesit folosind butonul " +
            e.getActionCommand());
        dispose(); // inchide fereastra curenta
    }
    public void windowClosing(ActionEvent ee) {
        System.out.println("Ati iesit in mod standard");
        dispose(); // inchide fereastra curenta
    }
}
// obligatoriu si celelalte metode ale interfeței WindowListener

```

Exemplul următor prezintă o metodă prin care putem evita scrierea simultană în două zone de text, folosind evenimentul `CaretEvent` generat în momentul schimbării poziției cursorului de componente text (derivate din `JTextComponent`).

```

JTextField tloc = new JTextField(); // prima componenta text
JTextField tcod = new JTextField(); // a doua componenta text
JLabel lloc = new JLabel();
JLabel lcod = new JLabel();
//... asculta prima zona de text
tloc.addCaretListener(new javax.swing.event.CaretListener() {
    public void caretUpdate(CaretEvent e) {
        tloc_caretUpdate(e);
    }
});
//... asculta a doua zona de text
tcod.addCaretListener(new javax.swing.event.CaretListener() {
    public void caretUpdate(CaretEvent e) {
        tcod_caretUpdate(e);
    }
});
// tratarea miscarii cursorului prin introducerea de continut
void tloc_caretUpdate(CaretEvent e) {
    int pozitie=e.getDot();
    if (pozitie>0)
        { tcod.setText(""); }
}
// tratarea miscarii cursorului prin introducerea de continut
void tcod_caretUpdate(CaretEvent e) {
    int pozitie=e.getDot();
    if (pozitie>0)
        { tloc.setText(""); }
}

```

Unei componente îi corespund un număr limitat de tipuri de evenimente pe care le poate genera prin interacțiunea cu utilizatorii. Pentru a le determina trebuie să vă uitați la ascultătorii pe care componenta îi poate înregistra (puteți folosi *java-doc-ul*). O componentă generează numai evenimentele pentru care are ascultători înregistrați. Spre exemplu, dacă unui câmp text (eng. *text field*) îi adăugați un singur ascultător care să trateze mișcările cursorului (eng. *caret*), atunci va arunca astfel de evenimente, dar nu va mai genera evenimente în momentul obținerii focusului (deținerea fluxului cu tastatura), apăsării butoanelor mouse-ului sau al oricărei alte acțiuni, deoarece nu are ascultători corespunzători asociați.

O dată ascultătorii adăugați unei componente, putem obține referințe despre ei folosind metoda pusă la dispoziție de clasa *JComponent*, al cărei prototip este

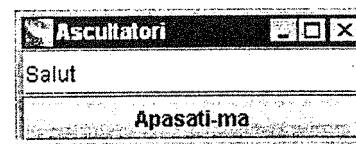
`EventListener[] getListeners(Class tipAscultator)`. Astfel, putem obține într-un tablou toți ascultătorii atașați componentei până în momentul curent. Parametrul *tipAscultator* reprezintă tipul ascultătorilor ce se doresc returnați, o instanță a clasei *Class* (a se vedea *Java Reflection API*) care poate fi expresia *Ascultator.class* dacă se vor a fi returnați ascultătorii de tip *Ascultator*. În exemplul următor vom adăuga un ascultător componentei, numai în cazul în care aceasta nu are deja un ascultător de același tip deja atașat.

```

JTextField tf=JTextField();
MouseListener un=new MouseListener()
{
    // suprascriere
};
//
MouseListener[] ml=(MouseListener[]) (c.getListeners(
    MouseListener.class));
if(ml.length==0)
    tf.addMouseListener(un);

```

Același eveniment poate fi generat de componente diferite. Spre exemplu, *Action Event* poate fi generat de componenta *JButton*, dar și de componenta *JTextField*. În cazul butonului, evenimentul este aruncat prin apăsarea lui cu mouse-ul sau prin apăsarea tastei *Space* când deține focusul în cazul look-and-feel-ului obișnuit. Un plugin look-and-feel poate implementa un buton care generează un astfel de eveniment atunci când utilizatorul pronunță o anumită frază. Pentru un câmp text, ceea ce determină generarea evenimentului este apăsarea tastei *Enter*. Dacă vom asocia un același ascultător ambelor componente, ne vom folosi de informațiile pe care evenimentul le poartă (în acest caz, sursa care l-a generat), pentru a obține o referință spre sursa evenimentului. Vom folosi metoda public *Object getSource()*, moștenită din clasa *EventObject* căreia îi aplicăm operatorul cast spre componenta dorită.



```
protected void faContinut() throws Exception
```

```

{
    JPanel continut= new JPanel();
    // in acest panou vom adauga continutul ferestrei
    continut.setLayout(new GridLayout(2,1));
    // am modificat gestionarul panoului continut
    this.getContentPane().add(continut);
    // am adaugat ferestrei panoul continut
    JTextField text = new JTextField();

```

```

        JButton buton= new JButton("Apasati-ma");
        continut.add(text);
        continut.add(buton); // am adaugat butonul
        Ascultator asc=new Ascultator(); // instantiem ascultator
        text.addActionListener(asc); // adaugam ascultator
        buton.addActionListener(asc);
    }

    class Ascultator implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            Object comp=e.getSource();
            // am obtinut componenta care a generat evenimentul
            if (comp.getClass().getName()=="javax.swing.JButton")
                System.out.println("Ati apasa butonul");
            else
                System.out.println("Ati terminat de scris");
                System.out.println("Informatii purtate de eveniment"
                    +e.toString());
        }
    }

```

Evenimentele pot fi împărțite în două grupe: evenimente de nivel scăzut (eng. *low level*) și evenimente semantice. Evenimentele de nivel scăzut sunt cele determinate de dispozitivele de intrare/ieșire, cum ar fi mouse-ul sau tastatura, schimbarea dimensiunii și poziționării componentelor, adăugarea sau ștergerea de componente din container, schimbarea focusului.

Evenimentele semantice sunt, spre exemplu, cele generate de introducerea unui caracter într-un câmp text, selectarea unui buton selectabil (eng. *check box*), alegerea unei opțiuni dintr-o listă deci, în general, interacțiuni directe ale utilizatorilor cu componentele atomice. Toate evenimentele folosite până acum în exemple sunt de acest tip.

Este de preferat atunci când se poate ca utilizatorii să înregistreze componentelor ascultători pentru evenimentele semantice pentru robustețea și portabilitatea aplicației. Spre exemplu, este de preferat să se asculte când un buton este apăsat decât să se capteze dacă butonul mouse-ului a fost apăsat în timpul în care cursorul se afla poziționat deasupra suprafeței butonului.

Prezentăm, în continuare, o serie de evenimente de nivel scăzut pe care le pot genera toate componentele din cauza descendenței lor din clasa Component din AWT:

Nume eveniment	Nume ascultător	Nume adaptor	Explicație eveniment
ComponentEvent	Component Listener	Component Adapter	Eveniment generat în momentul în care componenta își schimbă mărimea, poziția sau vizibilitatea.

Nume eveniment	Nume ascultător	Nume adaptor	Explicație eveniment
FocusEvent	FocusListener	FocusAdapter	Eveniment generat în momentul în care componenta capătă sau pierde posibilitatea de a primi comenzi de la tastatură.
KeyEvent	KeyListener	KeyAdapter	Eveniment generat de apăsarea unei taste doar de componentele care dețin focusul la acel moment.
MouseEvent	MouseListener	MouseAdapter	Eveniment generat de apăsarea butoanelor mouse-ului sau de intrarea sau ieșirea cursorului din suprafața componentei.
MouseMotionEvent	MouseMotionListener	MouseMotionAdapter	Eveniment generat de mișcarea mouse-ului în interiorul suprafeței componentei.

Ne vom concentra, în mod special, asupra evenimentelor generate de dispozitivele standard de intrare. Presupunem cazul unei componente `JTable` tabela care prin apăsarea de două ori a butonului drept va afișa o fereastră de autentificare, iar prin apăsarea butonului drept va afișa un meniu pop-up (meniu contextual), permițând operațiile clasice de editare:

gheorghe	vranceanu
teodor	panaitescu
dumitru	ghiu
ion	Copie Alt+C
nect	Taie Alt+T
cost	Lipeste Alt+V
nect	Sterge Alt+S
mihaela	
costache	dumbravanu

```

tabela.addMouseListener(new MouseAdapter() {
    // suprascrierea metodei care se va apela la apasarea
    // butonului mouse-ului
    public void mousePressed(MouseEvent e) {
        int i=e.getClickCount();
        // i pastreaza numarul de apasari
        if (i==2 && e.getButton()==1)
        {
            // cazul in care am apasat o data butonul din dreapta
            Autentificare protectie=new Autentificare(null,
                "Autentificare",true);
        }
        if (e.getButton()==3 && i==1 && editabila=true)
        {
            // cazul in care am apasat butonul stang o singura data

```

```

        pop.show(e.getComponent(), e.getX(), e.getY());
        // coordonatele curente ale mouse-ului se afla cu
        // metode de tipul get()
        pop.setVisible(true);
        // afisam meniul pop-up si il facem vizibil
    }
}
});

```

În cazul unui buton *Iesire* putem să adăugăm un ascultător care să trateze apăsarea tastei *Enter* atunci când el deține focusul (în look-and-feel-ul *Metal* apăsarea tastei *Enter*, când suntem poziționați pe buton, nu aruncă *ActionEvent*):

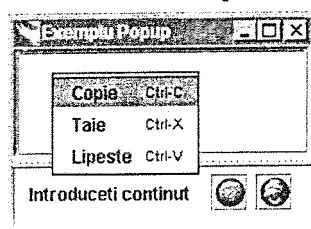
```

iesire.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        int i = e.getKeyChar(); //se determină codul tastei apăsate
        if (i==10) // daca am apasat tasta Enter
        {
            dispose();
        }
    }
});

```

Sistemul de ascultători și evenimente asociat unei componente este gestionat prin intermediul clasei `javax.swing.event.EventManagerList` care conține un tablou de perechi de forma `XXEvent/XXListener`. Câte o instanță a acestei clase este asociată fiecărei componente. Modelele implicite au la rândul lor asociate astfel de instanțe. Când utilizatorul înregistrează un ascultător unei componente, aceasta adaugă în tabloul `EventManagerList` instanța `Class` corespunzătoare evenimentului ce se vrea captat (a se vedea `Java Reflection API`). Următorul element introdus în tablou este chiar ascultătorul evenimentului, tabloul arătând astfel ca o succesiune de forma `XXEvent, XXListener, YYEvent, YYListener...` Atunci când un eveniment este generat, este parcurs în mod secvențial acest tablou până la găsirea unei potriviri și evenimentul este trimis ascultătorului corespunzător.

Putem crea ascultători complecși pentru componente cum ar fi cel care urmează, fiind adăugat unei componente text printr-un apel `addMouseListener(new PopUpSimplu())`, un ascultător determină apariția unui meniu contextual în care se pot face operații cu clipboard-ul sistemului de operare.



```

import javax.swing.*; // pentru clasa JPopupMenu
import java.awt.event.*; // pentru clasa MouseAdapter
import java.awt.*; // pentru clasa Point
import javax.swing.text.*; // pentru clasa JTextComponent

class PopUpSimplu extends MouseAdapter
{
    JPopupMenu pup=new JPopupMenu("Operatii");
    static JMenuItem copie;
    static JMenuItem taie;
    static JMenuItem lipeste;
    JTextComponent comp;

    public PopUpSimplu() {
        // cream optiunile meniului contextual care sunt instante
        // JMenuItem
        copie = new JMenuItem("Copie", new ImageIcon("c.gif"));
        // adaugam ascultator optiunii
        copie.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                comp.copy(); // copiem continut selectat
            }
        });
        pup.add(copie); // am adaugat optiunea meniului
        // adaugam acceleratori care vor aparea in dreapta
        // optiunii
        copie.setAccelerator(KeyStroke.getKeyStroke( KeyEvent.VK_C,
            ActionEvent.CTRL_MASK));

        taie= new JMenuItem("Taie", new ImageIcon("cu.gif"));
        taie.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                comp.cut(); // taiem continut selectat
            }
        });
        pup.add(taie);
        taie.setAccelerator(KeyStroke.getKeyStroke( KeyEvent.VK_X,
            ActionEvent.CTRL_MASK));

        lipeste= new JMenuItem("Lipește", new ImageIcon("p.gif"));
        lipeste.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                comp.paste(); // lipim zonei text continutul clipboardului
            }
        });
        pup.add(lipeste);
    }
}

```

```

lipeste.setAccelerator(KeyStroke.getKeyStroke(
KeyEvent.VK_V, ActionEvent.CTRL_MASK));
}

public void mousePressed(MouseEvent e) {
    int i=e.getClickCount(); // i tine numarul de apasari
    // pe buton
    comp=(JTextComponent)e.getComponent();
    // am obtinut o referinta spre componenta text
    if (i==1 && e.getButton()==3)// daca am apasat o
    // data butonul drept
    {
        Point punct=e.getComponent().getLocationOnScreen();
        // am determinat pozitia absoluta a cursorului mouse-ului
        pup.show(comp, e.getX(), e.getY());
        // am vizualizat pop-up
        pup.setVisible(true);
    }
}
}

```

13.3.2. Fire de execuție în Swing

Toate evenimentele sunt procesate de ascultători care le primesc în interiorul unui singur fir de execuție, care poartă numele de expeditor de evenimente (eng. *event-despatching thread*). Numele acestui fir de execuție provine de la faptul că el are grijă ca evenimentele să ajungă la ascultători pentru a fi tratate. De asemenea, toate desenările de componente, precum și schimbările dimensiunilor lor se vor petrece în acest fir de execuție. Event-despatching thread joacă un rol extrem de important în Swing și AWT și este cel care asigură controlul asupra schimbării stării și modului de vizualizare a componentelor într-un mod secvențial, nepermițând astfel efecte neașteptate.

Event-despatching thread are asociată o coadă (o structură de tipul FIFO – First In First Out) de evenimente care reprezintă de fapt coada de evenimente a sistemului. Orice eveniment care apare determină execuția codului care tratează evenimentul respectiv, generând modificarea proprietăților componente, a vizualizării sau redesenarea ei. Evenimentele adunate în coada de evenimente sunt procesate serial (unul după altul), pentru a evita situații de genul modificării stării componente în mijlocul redesenării, ceea ce determină efecte neașteptate. Cunoșcând acest lucru trebuie să avem grijă să nu expediem evenimente în afara firului de execuție event-despatching thread.

Cea mai mare problemă, în acest sens, o reprezintă modificări pe care le putem face asupra interfeței grafice după ce aceasta a fost realizată. Pentru un container de bază cum ar fi `JFrame`, `JDialog`, a fi realizat (eng. *realized*) înseamnă că le-a fost apelată metoda `setVisible(true)`, `show()` sau `pack()`. Când o componentă este

realizată, toate componentele pe care le conține se realizează. O altă metodă de a realiza o componentă este adăugarea ei într-un container deja realizat.

În general, pentru cele mai multe aplicații nu trebuie să purtăm grija firelor de execuție. Este de ajuns să fie respectat următorul șablon care asigură realizarea corectă a interfeței.

```

public static void main( String[] args ) {
    Fereastra fereastra = new Fereastra();
    //...
    // aici se adauga componente ferestrei daca nu am facut-o
    // deja in constructor
    fereastra.pack(); // aducem componentele la dimensiunea lor
    // preferata
    fereastra.setVisible(true);
    // nu se mai fac operatii asupra interfetei
}

```

Se știe că metoda `main()` este executată într-un fir de execuție startat la execuția aplicației care poartă numele *main thread*. Prin apelarea unor metode de genul `setVisible()`, `show()` sau `pack()`, controlul asupra interfeței trece din *main thread* spre *event-despatching thread*, unde este executat în siguranță.

Pentru a ne ajuta să executăm codul nostru doar din interiorul firului *event-despatching* (pentru a ne proteja de evenimente neplăcute), Swing oferă o clasă numită `SwingUtilities`. Această clasă conține două metode care ne interesează în acest context. Prima este `invokeLater()`, care permite execuția de cod direct în firul *event-despatching*. Codul ce se vrea executat trebuie așezat în interiorul metodei `run()` dintr-un obiect `Runnable` și acesta trebuie introdus ca argument pentru `invokeLater()`, așa cum se vede în exemplul următor:

```

Runnable deExecutat = new Runnable() {
    public void run() {
        faCeva(); // metoda care contine codul nostru
    }
};
SwingUtilities.invokeLater(deExecutat);

```

A doua metodă pe care clasa `SwingUtilities` o pune la dispoziție este `invokeAndWait()`, care diferă față de precedenta prin faptul că nu se redă controlul aplicației până nu se execută codul din metoda `run()`. Aveți în continuare un exemplu:

```

try {
    Runnable asteaptaExecutia = new Runnable() {
        public void run() {
            faCeva(); // codul ce se vrea executat
        }
    };
    SwingUtilities.invokeAndWait(asteaptaExecutia);
}

```

```

    }
    catch (InterruptedException ie) {
        System.out.println("...s-a intrerupt executia!");
    }
    catch (InvocationTargetException ite) {
        System.out.println("...eroare in metoda run()");
    }
}

```

Deoarece aceste bucăți de cod sunt plasate în coada de evenimente, ele se execută din interiorul firului event-despatching și trebuie să avem grijă ca ele să se execute rapid, ca orice metodă care tratează evenimente. Dacă metoda `faCeva()` din exemplele precedente are un timp lung de execuție, vom observa că aplicația îngheață. Pentru a rezolva astfel de probleme putem crea propriul nostru fir de execuție, care să realizeze diverse operații care au nevoie de timp, cum ar fi conectări la rețea, încărcări de imagini etc.:

```

Thread iaTimp = new Thread() {
    public void run() {
        faCevaIndelungat(); // o actiune care ia timp
        SwingUtilities.invokeLater( new Runnable () {
            public void run() {
                reDeseneazaComponentele(); // se reactualizeaza
                componentele
            }
        });
    }
};
iaTimp.start();

```

Referitor la firul event-despatching apare noțiunea „metode sigure” (eng. *thread safe*). Câteva metode din Swing sunt construite ca fiind *thread safe* și nu au nevoie de tratare specială. Acest lucru este, în general, anunțat în java-doc. Sunt câteva metode *thread safe* chiar dacă nu sunt marcate ca fiind așa, și anume: `repaint()`, `revalidate()` și `invalidate()` (a se vedea subcapitolul următor).

Pentru a executa cod numai în interiorul event-despatching thread putem folosi următorul șablon:

```

public void codSigur() {
    if (SwingUtilities.isEventDispatchThread()) {
        //
        // ceea ce trebuie facut...
        //
    }
    else {
        Runnable apeleazaCodSigur = new Runnable() {
            public void run() {
                codSigur();
            }
        };
        SwingUtilities.invokeLater(apeleazaCodSigur);
    }
}

```

```

    }
    };
    SwingUtilities.invokeLater(apeleazaCodSigur);
}

```

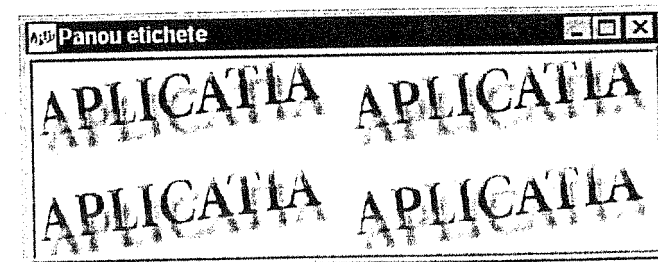
13.3.3. Desenarea componentelor grafice

Tot ceea ce este vizualizat pe ecran la un moment dat, în urma execuției unei aplicații având interfață grafică, rezultă în urma unui proces de desenare. Componentele sunt desenate. Procesul de desenare începe cu containerele de bază și se continuă cu conținutul. Componentele Swing se redesenează singure atunci când este nevoie, spre exemplu, atunci când schimbăm dimensiunile ferestrei sau când fereastra revine în prim-plan după ce a fost acoperită de o altă fereastră. Sau când apelăm pentru o etichetă o metodă de genul `setText()`, spre exemplu, eticheta se va redesena singură (conținând noul text) și se va și redimensiona pentru a-l cuprinde. Utilizatorul nu trebuie să poarte grija desenării decât în cazul în care dorește să creeze propriile lui componente sau să depaneze procesul de desenare.

În Swing, pentru ca o componentă să aibă acces la desenarea sa, este necesar să suprascrie metoda `paintComponent()`, spre deosebire de AWT, unde trebuia suprascrie `paint()`. În interiorul acestei metode avem acces la obiectul de tip `Graphics` pe care componenta îl conține, cunoscut și sub numele de contextul grafic al obiectului (eng. *graphic context*). Clasa `Graphics` conține multe metode care permit desenarea figurilor geometrice sau scrierea de text cu diferite fonturi, dimensiuni și culori.

Clasa `Graphics` folosește ceea ce se numește zona de lucru (eng. *clipping area*). În interiorul unei metode `paint()`, această zonă reprezintă regiunea din vizualizarea componentei care va fi redesenată. Mai poartă numele de regiunea murdară (eng. *dirtied*) a componentei și numai această zonă va fi reactualizată prin redesenarea componentei. Putem determina dimensiunile acestei regiuni folosind metoda `getBounds()` pe care o pune la dispoziție clasa `Graphics`. Suprafața de desenare se poate micșora din motive de eficiență, spre exemplu, nefiind necesară desenarea unei anumite regiuni la un moment dat.

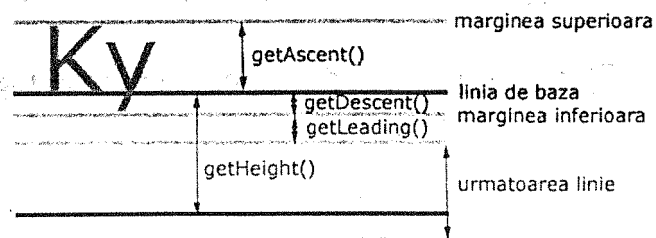
Prezentăm în continuare personalizarea unui panou care va conține o etichetă cu lățimea de 180 pixeli și înălțimea de 60 pixeli, care se repetă umplându-i conținutul:



```
// personalizăm clasa JPanel suprascriind metoda
paintComponent()
JPanel ptEtichete=new JPanel() {
    public void paintComponent(Graphics g) {
        Rectangle r=g.getClipBounds(); // zona desenabila
        int i=(int)r.getHeight(); // inaltimea zonei
        int l=(int)r.getWidth(); // latimea zonei
        int nri=i/60; // de cate ori incapa pe verticala
        int nrl=l/180; // de cate ori incapa pe orizontala
        for (int t=0;t<=nrl;t++)
            for (int n=0;n<=nri;n++)
                g.drawImage(img,t*180,n*60,null);
    }
};
```

În procesul de desenare, Swing folosește o tehnică care se numește *buffer dublu* (eng. *double-buffering*), care reprezintă desenarea într-o imagine, și nu desenarea direct într-o componentă vizibilă pe ecran. La finalul procesului de desenare imaginea este vizualizată pe ecran (proces care se petrece rapid). Spre deosebire, în AWT dezvoltatorii erau cei responsabili pentru implementarea acestui mecanism pentru a reduce flash-urile. Pentru a seta/deseta acest mecanism pentru o componentă Swing se folosește metoda public void setDoubleBuffered(boolean aFlag) din clasa JComponent. Dacă am setat buffer dublu pentru o anumită componentă, toate componentele conținute se vor desena folosind acest mecanism.

Vom prezenta, în continuare, metodele prin care contextul grafic (în fapt, clasa Graphics) permite desenarea textului cu diferite fonturi, dimensiuni, culori. Mai întâi, câte ceva despre modul în care este structurat textul în cazul modului grafic pentru Java. Reperul pentru desenarea fiecărei linii de text îl reprezintă o axă orizontală care poartă numele de linia de bază (eng. *base line*). Literele care nu se prelungesc în jos (spre exemplu, litera X) se desenează deasupra (eng. *ascent*) acestei linii de bază. Sub linia de bază rămâne să se deseneze partea de jos a literei (eng. *descent*) pentru diacritice precum ș sau ț ori pentru litere precum y. Înălțimea unui rând reprezintă distanța dintre două linii de bază consecutive. În figura care urmează sunt prezentate toate cele discutate.



Se observă faptul că extrema de sus a unui rând este marcată printr-o linie reprezentând marginea superioară (eng. *ascender line*), iar marginea de jos este delimitată

printr-o linie reprezentând marginea inferioară (eng. *descender line*). Desenarea efectivă a textului se face cu metoda drawString(String cuvânt, int x, int y) pe care clasa Graphics o pune la dispoziție. Precizăm că x și y sunt coordonatele colțului din stânga jos ale textului, și, mai precis, y reprezintă chiar linia de bază.

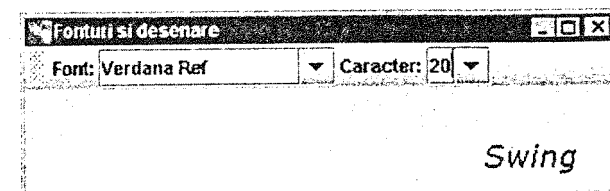
Înainte de a realiza operația de desenare trebuie să setăm fontul cu care se va realiza aceasta folosind metoda setFont(Font font) din clasa Graphics. Clasa java.awt.Font permite crearea de fonturi folosind constructori de forma public Font(String name, int style, int size) unde name reprezintă numele fontului, de exemplu, Verdana sau Tahoma, sau altul, style reprezintă stilul fontului care poate fi una din constantele: Font.BOLD, Font.ITALIC sau Font.PLAIN, iar size reprezintă mărimea caracterelor.

Pentru a obține caracteristicile fontului atribuit deja unui context grafic avem la dispoziție clasa FontMetrics. Instanța acestei clase asociată contextului grafic curent g se obține printr-un apel de forma FontMetrics fm=g.getFontMetrics(). Prezentăm metodele pe care clasa FontMetrics le pune la dispoziție pentru a obține caracteristicile fontului, necesare în procesul de poziționare a textului.

Prototipul metodei	Explicație
int getAscent()	Metoda getAscent returnează distanța în pixeli dintre linia ascender și linia de bază.
int getDescent()	Metoda getDescent returnează numărul de pixeli dintre linia de bază și linia descender.
int getHeight()	Returnează numărul de pixeli dintre linia de bază a unui rând și linia de bază a rândului următor.
int getLeading()	Returnează distanța dintre o linie de text și următoarea.

Culorile sunt gestionate în AWT și Swing prin clasa java.awt.Color care ne permite construcția de culori RGB prin constructori de forma: Color(int r, int g, int b), unde r, g, b reprezintă întregi de la 0 la 255. Clasa Color definește și o serie de constante care reprezintă culorile simple, ca, de exemplu, public static final Color black. Asocierea unei culori contextului grafic g se face printr-un apel de genul g.setColor(Color.black). După această asociere, metodele clasei Graphics vor desena cu respectiva culoare.

Prezentăm o mică aplicație în care se folosesc toate noțiunile discutate despre fonturi și culori. Aplicația permite setarea fonturilor pe care sistemul le pune la dispoziție și a dimensiunilor lor din componente ComboBox, care apoi sunt aplicate asupra textului „Swing”. Culoarea este setată semialeator folosind coordonatele obținute prin FontMetrics.




```

/* Fereastră aplicatiei */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class FereastrăAplicatie extends JFrame {
    //variabile de clasă
    static JMenuBar meniuBara;
    final Image img; // iconul ferestrei
    int pozitie=230;
    //constructorul clasei
    public FereastrăAplicatie() throws Exception {
        super("FereastrăAplica\u0163ie");
        // poziționăm fereastra
        Dimension marimeEcran =
            Toolkit.getDefaultToolkit().getScreenSize();
        setBounds ( pozitie, pozitie, marimeEcran.width-2*pozitie,
            marimeEcran.height/2-30);
        // setăm iconul ferestrei
        Toolkit t=this.getToolkit();
        img=t.getImage("icon.jpg");
        this.setIconImage(img);
        this.getContentPane().setLayout(new BorderLayout());
        faContinut();
    }
    Font f=new Font("Tahoma", Font.BOLD, 10); // un font nou
    protected void faContinut() throws Exception
    {
        // personalizăm un panou
        final JPanel continut= new JPanel()
        {
            public void paintComponent(Graphics g)
            {
                String sir="Swing";
                g.setFont(f);
                FontMetrics fm=g.getFontMetrics();
                // fm va conține caracteristicile fontului contextului
                // grafic
                int i=fm.getDescent();
                int k=fm.getAscent();
                System.out.println("i="+i+"k="+k);
                // cream o culoare nouă (nuante închise)
                Color culoare=new Color(i%255, k%255, i*k%255);
            }
        };
    }
}

```

```

// setăm culoarea contextului grafic
g.setColor(culoare);
// desenăm textul
g.drawString(sir, 300, 100);
    }
};

GraphicsEnvironment g=GraphicsEnvironment.
getLocalGraphicsEnvironment();
String[] fonturi =g.getAvailableFontFamilyNames();
// variabila fonturi conține fonturile sistemului de
// operare
final JComboBox fontBox = new JComboBox(fonturi); //
combobox fonturi
fontBox.setEditable(true); // se poate edita combobox-ul
fontBox.setSelectedItem("Tahoma"); // setăm fontul
combobox-ului
fontBox.setMaximumSize(fontBox.getPreferredSize()); //
dimensiunea minimă
String[] mar = {"10", "15", "20"};
final JComboBox mB = new JComboBox(mar);
// combobox marimi
mB.setMaximumSize(mB.getPreferredSize());
ActionListener ascFont = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
            public void run()
            {
                String str = fontBox.getSelectedItem().toString();
                int gr=Integer.parseInt(mB.getSelectedItem().toString());
                f=new Font(str,Font.ITALIC,gr);
                FereastrăAplicatie.this.repaint();
            }
        };
        runner.start();
    }
};

fontBox.addActionListener(ascFont);
mB.addActionListener(ascFont);
JToolBar tb = new JToolBar();
tb.add(new JLabel(" Font: "));
tb.add(fontBox);
fontBox.setEditable(true);
tb.add(new JLabel(" Caracter: "));
tb.add(mB);
mB.setEditable(true);

```

```

this.getContentPane().add(tb, BorderLayout.NORTH);
this.getContentPane().add(continut, BorderLayout.CENTER);
}
}

```

De asemenea, pe lângă text putem să desenăm diferite figuri geometrice, de la cele mai simple la cele mai complexe. Platforma Java 2 oferă un nou pachet pentru desenare, și anume Java2D, care permite următoarele: desenarea cu diverse stiluri de linii, umplerea figurilor geometrice cu texturi și gradienti, mutări, rotiri, scalări de text și figuri geometrice, suprapuneri de text și figuri, lucrul cu imagini, printarea.

13.3.4. Alinierea la JavaBeans

Trebuie remarcat faptul că, prin modul de construcție, orice componentă Swing este o componentă JavaBeans, acest lucru având importante implicații.

O definiție a componentelor ar putea fi următoarea: mici bucăți de soft compilate, reutilizabile și personalizabile, care pot fi încărcate într-un mediu de dezvoltare și integrate împreună cu alte componente pentru a forma o nouă aplicație, fără a fi recompilate. De asemenea, se poate vedea componenta ca o implementare a unei funcționalități de sine stătătoare. Din acest punct de vedere, funcționalitatea fiecărei componente poate fi extrem de variată. O componentă poate să aibă o reprezentare vizuală, așa cum este și cazul componentelor grafice din Swing. Poate, de asemenea, să nu aibă o reprezentare vizuală, astfel de exemple fiind parsere de XML, interogări SQL, baze de date etc. În Java, componentele poartă denumirea de JavaBeans, iar specificația JavaBeans pe care o puteți găsi la adresa www.javasoft.com/beans/docs/spec.html stă la baza construirii componentelor în Java.

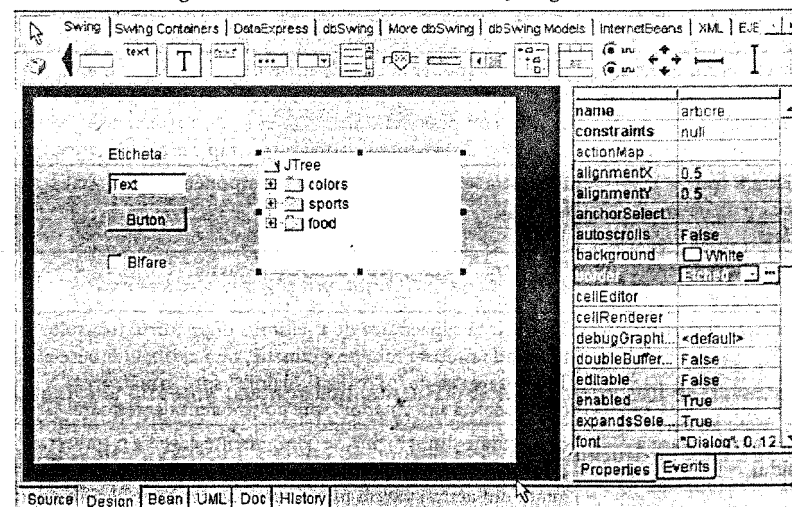
Componentele în Java sunt modelate folosind clasele. Astfel, componentele sunt clase care țin anumite proprietăți ce pot fi accesate numai prin metode de acces specificate. Proprietățile le putem gândi ca perechi atribut-valoare, metodele de acces permițând citirea sau modificarea valorii unui atribut. Proprietățile nu sunt altceva decât variabile globale ale clasei, iar dacă `NumeProprietate` reprezintă numele proprietății, metodele de acces sunt, prin convenție, metode având numele de forma `setNumeProprietate()`, `getNumeProprietate()`, `isNumeProprietate()`.

Proprietățile care nu aruncă evenimente în momentul în care li se schimbă valorile poartă numele de proprietăți simple. De asemenea, componentele pot arunca evenimente, care pot fi tratate prin listenerii care le ascultă, înainte sau după ce se modifică valoarea proprietăților lor. Astfel, o proprietate se numește legată (eng. *bound property*) dacă se va arunca un eveniment `PropertyChangeEvent` după ce își schimbă starea. Aceste evenimente sunt ascultate de listenerii `PropertyChangeListener` adăugați componentelor folosind metoda `addPropertyChangeListener()`. O proprietate se numește constrânsă (eng. *constrained property*) dacă aruncă un eveniment `PropertyChangeEvent` înainte de a-și schimba starea. Evenimentele `PropertyChangeEvent` poartă trei informații, și anume: numele proprietății, vechea valoare și noua valoare. Aceste evenimente sunt ascultate de listenerii `VetoableChangeListener` adăugați componentelor folosind metoda `addVetoableChangeListener()`. Aceste evenimente și ascultători sunt definite în pachetul `java.beans`. O mostră de tratare a evenimentelor

generate de schimbarea valorii unei proprietăți puteți găsi în exemplul din secțiunea dedicată componentei `JOptionPane`.

Respectând aceste convenții în scrierea componentelor, un mediu de dezvoltare (eng. *builder*) va cunoaște în momentul dezvoltării aplicației ce proprietăți are componenta, dacă acestea se pot numai citi sau se pot și modifica, și cum se pot citi și scrie acestea. Această proprietate a componentelor de a se face cunoscute poartă numele de introspecție. În fapt, nu există nici o clasă `java.beans.Bean` pe care o clasă trebuie să o extindă pentru a putea fi privită ca componentă, ci e necesar doar ca acea clasă să respecte convențiile din specificația JavaBeans pentru a fi considerată componentă.

Prin mediu de dezvoltare vom înțelege o unealtă (eng. *tool*) care este în fapt un mediu vizual în care componentele pot fi încărcate, personalizate și legate între ele pentru a forma împreună aplicații de sine stătătoare. Pentru Java există mai multe medii de dezvoltare (eng. *IDE Integrated Development Environment*), dintre care s-au distins câteva: JBuilder produs de Borland, Forte for Java produs de Sun Microsystems, JDeveloper produs de Oracle etc. Unul dintre beneficiile pe care îl aduc mediile integrate este posibilitatea creării rapide a interfețelor grafice prin simpla tragere a componentelor pe suprafața de lucru, urmată de personalizarea lor. Timpul necesar pentru crearea unei interfețe grafice se micșorează astfel de la câteva ore la câteva minute. Prezintă, în continuare, o captură din mediul vizual JBuilder pentru a ilustra fereastra design în care se construiesc interfețele grafice:



S-a constatat că este mai ușor să utilizezi o componentă creată de alții, personalizând-o și integrând-o cu alte componente deja existente, decât să scrii cod care să-i suplinească funcționalitatea. Un alt beneficiu al utilizării componentelor este acela că firmele mici pot produce componente cu diverse funcționalități, care vor fi cumpărate de alții și utilizate în aplicațiile lor. Pe Web puteți găsi componente grafice gata făcute la următoarele adrese web: www.componentsource.com/Marketplace/ și industry.java.sun.com/solutions.

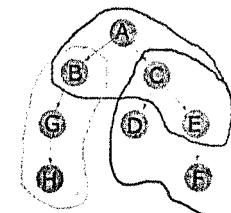
13.3.5. Gestiunea „focusului”

Prin focus vom înțelege proprietatea unei componente de a deține fluxul de intrare-ieșire cu tastatura. Prin așezarea componentelor într-un container Swing, ruta focusului este implicit de la stânga la dreapta și de sus în jos. Gestionarea programatică a focusului se realizează în mai vechile versiuni ale pachetului AWT (care reprezintă fundamentul pentru Swing) prin intermediul unei clase de serviciu care poartă numele de `FocusManager`. Din cauza proastei implementări a controlului focusului, începând cu distribuția J2SDK 1.4, rolul acestei clase a fost preluat de un alt gestionar numit `KeyboardFocusManager`, nerenunțându-se totuși la primul pentru compatibilitate. S-a încercat, de asemenea, prin acest nou gestionar o implementare pentru focus cât mai independentă de sistemul de operare peste care este instalată mașina virtuală. Atunci când veți dori să tratați focusul pentru interfața dumneavoastră, este de preferat să utilizați acest ultim manager a cărui specificație o puteți găsi chiar în `java-doc`, în directorul `\api\java\awt\doc-files\FocusSpec.html`.

Mai întâi trebuie înțelese câteva noțiuni, cu precizarea că sistemul de focus este gândit pentru AWT și valabil și pentru Swing:

Noțiuni	Explicații
Deținător al focusului (eng. <i>focus owner</i>)	Componenta care deține canalul de comunicare cu tastatura.
Deținător permanent al focusului (eng. <i>permanent focus owner</i>)	Este o componentă care deține focusul permanent. Spre exemplu, focus nepermanent au meniurile, bara de derulare dintr-o componentă <code>JScrollPane</code> , care atunci dețin focusul temporar doar pentru perioada de timp când sunt active.
Fereastra Window focusată (eng. <i>focused Window</i>)	Fereastra Window care conține componenta care deține focusul.
Fereastra activă (eng. <i>active Window</i>)	Fereastra <code>JFrame</code> sau <code>JDialog</code> focusată.
Traversarea focusului	Reprezintă capacitatea de a schimba deținătorul focusului folosind tastatura sau, programatic, fără ajutorul mouse-ului. Traversarea se poate realiza „înainte” spre următoarea componentă sau „înapoi” spre componenta anterioară.
Ciclul de traversare a focusului	O porțiune din ierarhia de componente care va fi traversată „înainte” sau „înapoi” într-un ciclu închis. Nici o altă componentă din afara ciclului nu va putea căpăta focusul prin operații „înainte”, „înapoi”.
Rădăcina ciclului focusului	Containerul care se află în vârful ierarhiei pentru un ciclu focus dat. Atunci când focusul este deținut de o componentă din interiorul unui ciclu dat, operațiile normale „înainte” și „înapoi” nu pot determina ieșirea focusului din ciclul curent. Pentru a realiza acest lucru se definesc alte două operații, „ciclul sus” și „ciclul jos”, care permit trecerea folosind tastatura sau programatic sus sau jos în ierarhia de cicluri de traversare a focusului.

Orice fereastră obișnuită sau interioară este implicit rădăcină a unui ciclu focus. Putem privi ierarhia arborescentă de componente care alcătuiește aplicația ca fiind partajată în diverse cicluri focus care au la rândul lor o așezare arborescentă, așa cum se poate vedea în figura care urmează:



Atunci când o componentă ajunge să dețină focusul sau îl pierde, este aruncat un eveniment `FocusEvent` care poate fi ascultat de un ascultător care implementează interfața `FocusListener`. Prezentăm un exemplu în care un buton își schimbă culoarea textului în roșu atunci când obține focusul și în negru atunci când îl pierde:

```
final JButton anuleaza = new JButton("Anuleaz\u0103");
anuleaza.addFocusListener(new FocusListener()
{
    public void focusGained(FocusEvent e)
    {
        // tratam obtinerea focusului
        anuleaza.setForeground(Color.red);
    }
    public void focusLost(FocusEvent e)
    {
        // tratam pierderea focusului
        anuleaza.setForeground(Color.black);
    }
});
```

Ordinea în care sunt parcurse componentele ciclului în procesul de traversare este determinată de diverse „politici” ce sunt clase care extind `FocusTraversalPolicy` și care se atașează rădăcinii ciclului de transfer al focusului printr-un apel `setFocusTraversalPolicy(FocusTraversalPolicy politica)`. O enumerare a implementărilor implicite care sunt puse la dispoziție este prezentată în continuare.

Tip „politică”	Explicație
<code>ContainerOrderFocusTraversalPolicy</code>	Traversarea ciclului se face în ordinea în care componentele au fost adăugate în container.
<code>DefaultFocusTraversalPolicy</code>	Traversarea ciclului se va face în aceeași ordine cu traversarea componentelor peer corespondente din sistemul de operare peste care este instalată mașina virtuală.

Tip „politică”	Explicație
SortingFocusTraversalPolicy	Traversarea ciclului se realizează în ordinea dată în urma sortării componentelor după un anumit criteriu dat de un obiect instanță al clasei Comparator.
LayoutFocusTraversalPolicy	Subclasă a clasei SortingFocusTraversalPolicy în care sortarea componentelor este făcută în funcție de mărime, poziție și orientare. Componentele sunt grupate după criteriile enunțate în linii și coloane, și traversarea se face pe linii de la stânga la dreapta și pe coloane, de sus în jos.

Pentru interfețele Swing care folosesc look-and-feel-urile standard sau un look-and-feel derivat din BasicLookAndFeel, politica de traversare implicită pentru containere este LayoutFocusTraversalPolicy. Interfețele AWT folosesc implicit DefaultFocusTraversalPolicy.

Trecerea de la o componentă la alta se realizează pentru sistemele de operare Windows și Unix implicit prin apăsarea tastei Tab sau a combinației Ctrl+Tab. Pentru a merge în sens invers va trebui să folosim implicit combinația de taste Shift+Tab sau Ctrl+Shift+Tab. Putem modifica tastele implicite pentru focus folosind un apel de genul setFocusTraversalKeys(int id, Set keystrokes). De asemenea, putem anula efectul apăsării tastelor pentru mutarea focusului folosind metoda setFocusTraversalKeysEnabled(boolean valoareAdevar).

Programatic avem acces la controlul fluxului prin intermediul metodelor următoare. Dacă nu au un parametru de intrare atunci au efect asupra componentei care deține focusul în momentul apelării lor.

Metoda	Descriere
focusNextComponent()	Trece focusul asupra componentei următoare.
focusPreviousComponent()	Trece focusul asupra componentei anterioare.
upFocusCycle()	Trece focusul asupra ciclului ascendent.
downFocusCycle()	Trece focusul asupra ciclului descendent.
focusNextComponent(Component c)	Trece focusul asupra componentei c.
focusPreviousComponent(Component c)	Trece focusul asupra componentei c.
upFocusCycle(Component c)	Trece focusul asupra componentei c.
downFocusCycle(Container c)	Trece focusul asupra containerului c.

Dacă dorim să transferăm focusul de la o componentă la următoarea sau precedentă din ciclu, putem folosi unul din apelurile transferFocus() sau

transferFocusBackward(). În exemplul următor, pentru o componentă facem ca apăsarea tastei tab să producă pierderea focusului și nu scrierea unui tab ca text.

```
//...
JTextArea text=new JTextArea();
// adaugam componentei text ascultatorul pentru apasările tastei
text.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        if (e.getKeyChar()==KeyEvent.VK_TAB) // daca s-a apasat Tab
        {
            if (e.isShiftDown()) // si este si tasta Shift apasata
            {
                // transferam focusul componentei precedente
                // (complementara)
                Continut.this.transferFocusBackward();
            }
            else
            {
                // transferam focusul componentei urmatoare din ciclu
                Continut.this.transferFocus();
            }
        }
    }
});
```

13.3.6. Clasificarea componentelor Swing

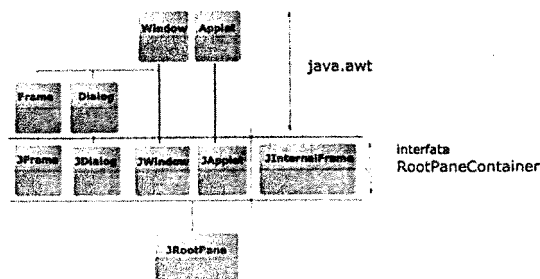
Componentele Swing pot fi împărțite în trei categorii, și anume: containere de bază, containere intermediare și componente atomice. Diferențele dintre ele le găsiți în tabelul următor:

Tip componentă	Explicații	Componente
Containere de bază	Orice aplicație care deține o interfață grafică are cel puțin un astfel de container. Componentele (care nu pot fi tot containere de bază) se vor adăuga acestui container. Reprezintă locul în care toate celelalte componente se vor desena.	JFrame, JWindow, JDialog, JApplet
Containere intermediare	Scopul lor este acela de a grupa mai multe componente și de a simplifica poziționarea, dimensiunile și modul în care componentele conținute vor reacționa la modificarea dimensiunilor ferestrei de bază. Pentru aceasta li se atașează gestionari de poziționare (eng. <i>layout manager</i>).	JPanel, JScrollPane, JSplitPane, JTabbedPane

Tip componentă	Explicații	Componente
Componente atomice	Nu au, precum celelalte tipuri de componente, rolul de a ține alte componente, ci sunt suficiente pentru a prezenta informația pe care o dețin utilizatorului sau pentru a primi informații de la utilizatorii interfeței grafice. Mai pe scurt, își sunt lor suficiente pentru a interacționa cu utilizatorul.	JLabel, JButton, JList, JComboBox, JTextField, JTextArea, JTable, JTree, JMenu, JPopup, JMenuItem, JToolBar, JOptionPane, JFileChooser, JColorChooser, JSlider, JScrollBar, JProgressBar, JDesktopPane, JInternalFrame.

13.4. Containere de bază

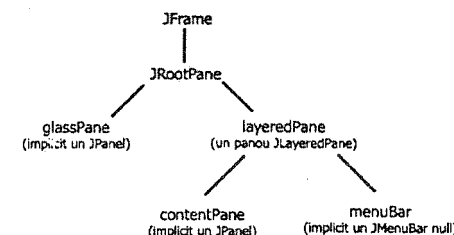
Toate cele 4 containere de bază implementează interfața `RootPaneContainer` și din acest motiv au aceeași organizare internă. Diferența dintre aceste containere și celelalte componente Swing este faptul că ele moștesc direct clasele corespunzătoare lor din AWT, așa cum se poate observa în figura care urmează. Toate celelalte componente moștesc clasa `JComponent`, și nu componentele corespunzătoare din AWT. Din acest motiv, ele sunt singurele componente „grele” din Swing (eng. *heavyweight*), fiind indirect dependente de componentele *peer* native ale sistemului de operare peste care este instalată platforma Java. Totuși, Swing oferă independența față de sistemul de operare de pe mașina gazdă (în sensul look-and-feel-ului componentelor care rămâne identic pentru toate sistemele și mașinile) tocmai prin modul în care aceste containere sunt construite, conținutul lor fiind gestionat de o componentă panou „ușoară”, și anume o instanță `JRootPane`, așa cum se va observa pe parcursul acestui capitol. Pentru alte amănunte despre diferențele dintre Swing și AWT, a se vedea subcapitolul AWT *vs.* Swing.



Vom insista pe componenta `JFrame`, iar la celelalte componente `JWindow`, `JDialog`, `JApplet` vom evidenția doar diferențele care apar. De asemenea, vom specifica și clasele care interacționează cu aceste containere.

13.4.1. JFrame

Containerul cel mai des folosit pentru o aplicație bazată pe Swing este `JFrame`. Modul în care această componentă este construită este mult mai complicat decât cel al componentei corespunzătoare din AWT, și anume `Frame`. Orice fereastră `JFrame` conține un câmp protejată având numele `rootPane`, care este o instanță a clasei `JRootPane`, servind drept container pentru alte câteva componente panou. Nu putem adăuga componente direct unei ferestre `JFrame` (așa cum procedam pentru o componentă `Frame`), ci unuia dintre panourile conținute în `rootPane` (nu lui `rootPane`, care este doar un panou intermediar). Aveți descrisă, în continuare, ierarhia implicită bazată pe relația *a part of* pentru o fereastră `JFrame`.



După cum se poate observa, obiectul de tip `RootPane` conține un câmp numit `glassPane` care este implicit un `JPanel` și un câmp `layeredPane` care este o instanță a clasei `JLayeredPane`. Acesta din urmă conține, la rândul lui, un câmp `contentPane`, implicit instanță a clasei `JPanel`, și câmpul `menuBar` de tip `JMenuBar`, care implicit este `null`.

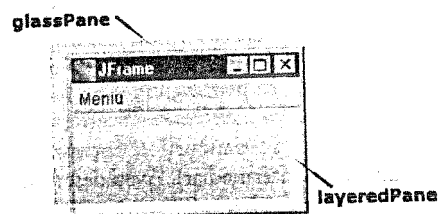
Pentru a adăuga o componentă unei ferestre `JFrame` trebuie, de fapt, să adăugăm acea componentă panoului `contentPane` (putem gândi acest panou ca gestionând conținutul ferestrei) și acest lucru se realizează prin apeluri de genul: `fereastrăMea.getContentPane().add(componentaMea)`. De asemenea, putem folosi un apel: `JPanel continut = (JPanel) getContentPane()`, obținând astfel panoul conținut, căruia putem să-i adăugăm componente folosind apeluri de genul `add()`.

Pentru a seta gestionarul de poziționare pentru o fereastră vom folosi un apel de forma: `fereastrăMea.getContentPane().setLayout(new FlowLayout())`.

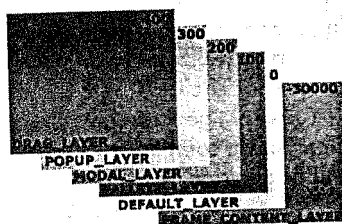
Dacă vom încerca să folosim metode de tip `add()` sau `setLayout()` direct pentru fereastra `JFrame` vom obține eroare. De remarcat metoda `getContentPane()` pe care clasa `JFrame` o pune la dispoziție pentru obținerea unei referințe la atributul `contentPane` asociat ferestrei.

Componenta `glassPane` este implicit un panou `JPanel` neopac care stă deasupra tuturor componentelor din `JRootPane`, acționând ca un ecran de protecție pentru fereastră. Acest lucru permite desenarea deasupra tuturor componentelor din fereastra curentă, precum și interceptarea sau întreruperea evenimentelor generate de mouse. Pentru a schimba `glassPane` din `JPanel` în oricare altă componentă folosim metoda public void `setGlassPane(Component glassPane)` pe care clasa `JFrame` o

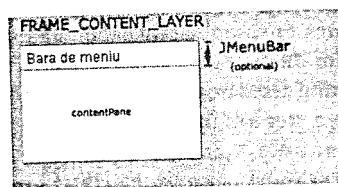
pune la dispoziție. O referință spre componenta `glassPane` curentă asociată unei ferestre se obține prin metoda public `Component getGlassPane()` tot din clasa `JFrame`. Implicit, `glassPane` care acoperă `layeredPane` nu este vizibilă, dar poate fi prin apelul `getGlassPane().setVisible(true)`.



`LayeredPane` este una dintre cele mai puternice și mai robuste componente din pachetul Swing. Este un container care oferă o a treia dimensiune pentru a poziționa componente, și anume adâncimea (eng. *Z-order*). Este un container având un număr nelimitat de straturi (eng. *layers*) pe care componentele pot sta. Pe un același strat putem așeza în același timp un număr nelimitat de componente. Atunci când adăugăm o componentă unui obiect de tip `LayeredPane`, trebuie să specificăm și adâncimea sub forma unui întreg care reprezintă numărul stratului. Cu cât numărul este mai mare, cu atât adâncimea este mai mică. Apare în acest fel fenomenul de suprapunere a componentelor conținute, acest lucru putând fi foarte util. Spre exemplu, dorim ca un meniu pop-up să apară deasupra celorlalte componente conținute de fereastră. Clasa `JLayeredPane` oferă 6 obiecte `Integer` constante reprezentând cele mai utilizate straturi, având următoarele nume: `FRAME_CONTENT_LAYER`, `DEFAULT_LAYER`, `PALETTE_LAYER`, `MODAL_LAYER`, `POPUP_LAYER` și `DRAG_LAYER`. În figura următoare puteți să vedeți și adâncimile straturilor menționate:



Stratul `FRAME_CONTENT_LAYER` este locul în care panoul rădăcină asociat fiecărei ferestre adaugă panoul conținut (`ContentPane`) și opțional bara de meniu. Îl prezentăm în figura următoare:

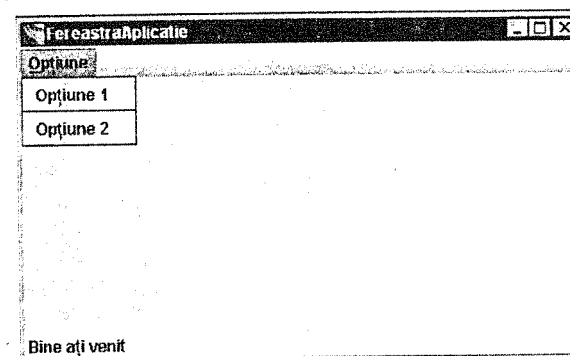


Bara de meniu nu există implicit, dar poate fi setată folosind metoda `setMenuBar()` din clasa `JFrame`:

```
JMenuBar meniu = new JMenuBar();
setMenuBar(meniu);
```

Astfel, bara de meniu este poziționată în partea de sus a stratului `FRAME_CONTENT_LAYER`, restul acestui strat fiind ocupat de panoul `ContentPane`, așa cum se observă în figura de mai sus. Implicit, `ContentPane` este un `JPanel` opac. Poate fi setat ca fiind oricare altă componentă printr-un apel de metodă aflată în clasa `JFrame`, și anume `setContentPane(componentaMea)`.

Prezentăm în continuare o clasă care construiește o fereastră și pe care o puteți folosi ca șablon (eng. *pattern*) pentru ferestrele aplicațiilor dumneavoastră, adaptând-o după necesități:



```
/* Fereastră aplicației */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class FereastrăAplicatie extends JFrame {
    //variabile de clasă
    static JMenuBar meniuBara;
    Image img;
    int pozitie=230;
    //constructorul
    public FereastrăAplicatie() throws Exception {
        super("FereastrăAplica\u0163ie");
        // poziționăm fereastra și-i setăm dimensiunile
        Toolkit t=this.getToolkit();
        Dimension marimeEcran = Toolkit.getDefaultToolkit().
            getScreenSize();
```

```

        setBounds ( pozitie, pozitie, marimeEcran.width-2*
        pozitie, marimeEcran.height/2-30);
        // setam iconul din coltul din stanga sus al ferestrei
        img=t.getImage("icon.jpg");
        this.setIconImage(img);
        this.getContentPane().setLayout(new BorderLayout());
        // construim partile componente
        faMeniu();
        faContinut();
        faBaraStare();
        // specificam modul de reactie al ferestrei la
        inchidere (e paraseste aplicatia)
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                iesire();
            }
        });

protected void faMeniu() {
    meniuBara = new JMenuBar(); // cream bara de meniu
    meniuBara.setOpaque(true);
    JMenu optiuni= faMeniuOptiuni(); //cream un meniu
    // optiuni
    optiuni.setMnemonic('O'); // definim shortcut (Alt+O)
    meniuBara.add(optiuni); // adaugam meniul optiuni la
    // bara de meniu
    setJMenuBar(meniuBara);
}

// construim meniul optiuni
protected JMenu faMeniuOptiuni() {
    JMenu unelte= new JMenu("Op\u0163iune");
    JMenuItem o1 = new JMenuItem("Op\u0163iune 1");
    JMenuItem o2 = new JMenuItem("Op\u0163iune 2");
    // adaugam ascultatorul pentru optiunea1
    o1.addActionListener(new ActionListener() {
        public void actionPerformed
        (ActionEvent e) {
            // actiunea in cazul alegerii optiunii 1
        }
    });
    // adaugam ascultatorul pentru optiunea2
    o2.addActionListener(new ActionListener() {
        public void actionPerformed
        (ActionEvent e) {
            // actiunea in cazul alegerii optiunii 2
        }
    });
}

```

```

        // adaugam optiunile la meniu
        unelte.add(o1);
        unelte.addSeparator();
        unelte.add(o2);
        return unelte;
    }

    // construim continut
    protected void faContinut() throws Exception
    {
        JPanel continut= new JPanel(); // in acest panou vom
        adauga continutul ferestrei
        this.getContentPane().add(continut, BorderLayout.CENTER);
    }

    protected void faBaraStare()
    {
        JPanel ajutor= new JPanel();
        // panoul care va tine eticheta de stare
        ajutor.setBorder(new SoftBevelBorder(
        SoftBevelBorder.RAISED));
        ajutor.setLayout(new BorderLayout());
        // punem in evidenta panoul prin bordura
        JLabel stare=new JLabel("Bine a\u0163i venit");
        // setam textul etichetei
        ajutor.add(stare, BorderLayout.CENTER);
        this.getContentPane().add(ajutor, BorderLayout.SOUTH);
        // adaugam bara de stare
    }

    public void iesire() {
        System.exit(0); // parasim aplicatia
    }
}

```

În general, o aplicație are o singură fereastră de bază care este apelată din interiorul clasei principale (cea care conține metoda main()), așa cum rezultă din codul următor:

```

/*Aplicatia care activeaza fereastra*/
import javax.swing.*;
public class Aplicatia {
    public static void main( String[] args ) {
        try
        {
            UIManager.setLookAndFeel(
                "com.incorps.plaf.kunststoff.KunststoffLookAndFeel");
            FereastraAplicatie fereastra = new FereastraAplicatie();

```

```

fereastra.setVisible(true);
}
catch(Exception ex)
{
    System.out.println("Eroare"+ex.toString());
}
}
}

```

13.4.2. Clase și interfețe legate de ferestre

Interfața `WindowConstants` este folosită pentru a specifica modul în care o fereastră `JFrame` sau `JDialog` reacționează la închidere folosind metoda `setDefaultCloseOperation(int constanta)`. Definește patru constante întregi a căror descriere o aveți în tabelul următor:

Constanta	Descriere
<code>DISPOSE_ON_CLOSE</code>	Distruge fereastra și conținutul.
<code>DO_NOTHING_ON_CLOSE</code>	Nu reacționează la operația de închidere.
<code>EXIT_ON_CLOSE</code>	Se părăsește și aplicația.
<code>HIDE_ON_CLOSE</code>	Ascunde fereastra, ea fiind ușor revizualizată la un moment ulterior.

Interfața `WindowListener` trebuie implementată de toți ascultătorii care tratează evenimentele produse de fereastra căreia i-au fost atașați folosind metoda public `void addWindowListener(WindowListener l)`. Această interfață are 7 metode, fiecare tratând un anumit tip de eveniment, toate trebuind suprascrise de către cei ce creează ascultători. Le prezentăm în continuare fără explicații, rolul lor înțelegându-se din denumirea pe care o poartă:

```

void windowActivated(WindowEvent e)
void windowClosed(WindowEvent e)
void windowClosing(WindowEvent e)
void windowDeactivated(WindowEvent e)
void windowDeiconified(WindowEvent e)
void windowIconified(WindowEvent e)
void windowOpened(WindowEvent e)

```

Clasa `WindowAdapter` este o implementare abstractă a interfeței `WindowListener` permițând prin extindere suprascrierea unui număr limitat de metode ale interfeței `WindowListener`.

Clasa `WindowEvent` reprezintă evenimentul generat în momentul schimbării stării ferestrei (închidere, minimizare...). Este captat și tratat de orice `WindowListener` sau `WindowAdapter` atașat ferestrei tocmai în acest scop. Există șapte

tipuri de evenimente, fiecare fiind tratat de metoda corespunzătoare din ascultător: `WINDOW_ACTIVATED`, `WINDOW_CLOSED`, `WINDOW_CLOSING`, `WINDOW_DEACTIVATED`, `WINDOW_DEICONIFIED`, `WINDOW_ICONIFIED`, `WINDOW_OPENED`. Prezentăm două metode ale acestor clase:

Metoda	Explicație
<code>public Window getWindow()</code>	Returnează fereastra sursă a evenimentului.
<code>public String paramString()</code>	Returnează un șir de caractere care descrie tipul de eveniment, sursa și altele; este util în procesul de depanare.

Clasa `Toolkit` permite obținerea rezoluției ecranului printr-un apel de forma: `Dimension dim = getToolkit().getScreenSize()`. Implicit, fereastra se va afișa în colțul din stânga sus al ecranului. Putem face ca fereastra curentă să apară în centrul ecranului, indiferent de rezoluția lui, folosind un apel de forma:

```

this.setLocation(dim.width/2-this.getWidth()/2, dim.height/
2-this.getHeight()/2);

```

Aceste clase și interfețe le veți găsi folosite și comentate în exemplele acestui capitol.

13.4.3. JWindow

Componenta `JWindow` este asemănătoare cu `JFrame`, cu excepția faptului că nu are bară de titlu, nu i se pot modifica dimensiunile, nu se poate minimiza, maximiza și nu poate fi închisă. `JWindow` poate fi folosită pentru a vizualiza un mesaj temporar sau logourile de la începutul utilizării aplicației. Deoarece implementează interfața `RootPaneContainer`, necesită același comportament din partea dezvoltatorului pentru a-i accesa conținutul. Prezentăm, în continuare, un exemplu de logo realizat folosind `JWindow` și vizualizat la începutul startării aplicației, folosind un fir de execuție:



```

import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

class Logo
{
    JWindow w;

```



```

public Logo(JFrame f)
{
    JWindow j=new JWindow(f);
    w=j;
    // inconjuram fereastra JWindow cu borduri pentru a nu se
    //intrepatrunde de fundalul
    Border raisedbevel =
        BorderFactory.createRaisedBevelBorder();
    Border loweredbevel =
        BorderFactory.createLoweredBevelBorder();
    Border compus = BorderFactory.
        createCompoundBorder(raisedbevel, loweredbevel);
    JPanel panu=new JPanel(); // panoul continut
    panu.setBorder(compus);
    panu.setBackground(Color.white);
    // imaginea
    ImageIcon imaginea = new ImageIcon("logo.jpg");
    //eticheta1
    Font font1=new Font("Comic Sans MS",Font.ITALIC, 40);
    JLabel eticheta1 = new JLabel("Aplica\u0163ia");
    eticheta1.setFont(font1);
    //eticheta2
    Font font2=new Font("Verdana",Font.BOLD, 20);
    JLabel eticheta2 = new JLabel("Versiunea 1.3");
    eticheta2.setFont(font2);
    panu.add(eticheta1);
    panu.add(new JLabel(imaginea));
    panu.add(eticheta2);
    j.getContentPane().add(panu);
    // stabilim dimensiunile si pozitionarea logo-ului
    Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();
    j.setLocation(screenSize.width/2-100,screenSize.height/2-75);
    j.setSize(200,150); //dimensiuni
    j.setBackground(Color.white);
    j.show();
    j.toFront();
    j.setVisible(true);
}

// metoda care elibereaza resursele alocate ferestrei
public void stop()
{
    w.dispose();
}
}

```

Vom activa fereastra logo printr-un fir de execuție. Sugerăm utilizarea unor modalități mai sofisticate pentru gestionarea timpului de expunere, ca de exemplu Timer.

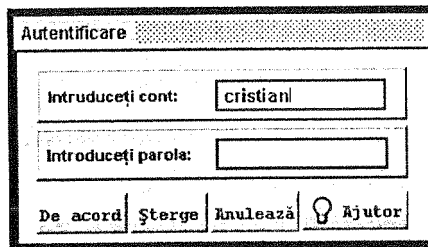
```

stoper=true;
Logo d;
//....
Thread fir1=new Thread()
{
    public void run()
    {
        d=new Logo(this);
        try {
            while(stoper)
            {
                sleep(100);
            }
            sleep(6000);
        }
        catch (Exception exex)
        {
            System.out.println("Eroare"+exex.toString());
            d.stop();
            stop(); // elibereaza efectiv resursele acestui
            thred
        }
    }
};
//.... dupa ce s-a desenat intreaga interfata grafica,
stergem logo-ul
stoper=false;

```

13.4.4. JDialog

Această clasă este folosită pentru a crea o fereastră de dialog ale cărei structură și comportament sunt identice cu `JFrame`. Un dialog permite realizarea interacțiunii utilizatorilor cu aplicația în timpul folosirii ei, permițând introducerea și obținerea de informații. Prin diversitatea de constructori oferiți putem crea o fereastră `JDialog` modală sau nemodală, dependentă sau nu de un alt container. Dacă este modală, nu se permite altor ferestre să fie active în același timp cu ea. Dacă este dependentă de un părinte dialogul se va activa doar în strânsă legătură cu el, altfel aplicația poate să aibă drept interfață grafică un simplu dialog și nimic altceva. Puteți consulta `java-doc-ul` pentru lista de constructori a componentei `JDialog`. Platforma Java mai pune la dispoziție o componentă pentru realizarea de dialoguri mai puțin complexe, și anume `JOptionPane`, care nu face subiectul acestui capitol. Prezentăm în continuare o clasă care folosește un dialog modal dependent de un părinte, prin derivare din clasa `JDialog`. Prezentăm codul care generează următoarea fereastră de dialog fără partea de formatare a componentelor conținute și tratarea evenimentelor ce pot apărea (pe acestea le găsiți în capitolele care descriu aceste subiecte):



```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;

public class Autentificare extends JDialog {
    JPanel panou = new JPanel();
    JLabel parole = new JLabel ();
    JLabel conte = new JLabel();
    JTextField contt = new JTextField();
    JPasswordField parolt = new JPasswordField();
    JButton deacord = new JButton();
    JButton sterge = new JButton();
    JButton anuleaza = new JButton();
    JButton ajutor = new JButton();

    public Autentificare(Frame frame, String title, boolean
    modal) {
        super(frame, title, modal);
        try {
            Initializare();
            int pozitie=20;
            Dimension marimeEcran = Toolkit.getDefaultToolkit().
            getScreenSize();
            setBounds (marimeEcran.width/2-150, marimeEcran.height/
            2-100,300, 170);
            setVisible(true);
            parolt.requestFocusInWindow();
            pack();
        }
        catch(Exception ex) {    ex.printStackTrace();    }
    }

    void Initializare() throws Exception {
        // desenam butoanele
        panou.setMaximumSize(new Dimension(350, 400));
```

```
panou.setPreferredSize(new Dimension(300, 200));
panou.setLayout(null);
contt.setText("cristian");
contt.setBounds(new Rectangle(139, 22, 119, 22));
parolt.setBounds(new Rectangle(139, 63, 120, 22));
deacord.setText("De acord");
deacord.setBounds(new Rectangle(14, 101, 63, 28));
deacord.setSelected(true);
sterge.setText("\u015Eterge");
sterge.setBounds(new Rectangle(79, 101, 54, 28));
anuleaza.setText("Anuleaz\u0103");
anuleaza.setBounds(new Rectangle(135, 101, 62, 28));
ajutor.setText("Ajutor");
ajutor.setBounds(new Rectangle(199, 101, 74, 27));
parole.setText(" Introduce\u0163i parola:");
parole.setBounds(new Rectangle(14, 55, 259, 37));
conte.setText(" Introduce\u0163i cont:");
conte.setBounds(new Rectangle(15, 14, 258, 37));
// aici urmeaza formatarea butoanelor si a zonelor de text
// si de asemenea tratarea evenimentelor
panou.add(parole, null);
panou.add(deacord, null);
panou.add(conte, null);
panou.add(sterge, null);
panou.add(anuleaza, null);
panou.add(ajutor, null);
panou.add(parolt, null);
panou.add(contt, null);
this.getContentPane().add(panou, null);
}
```

Pentru a obține o fereastră de dialog de genul celei de mai sus este nevoie de un apel de genul:

```
Autentificare protectie=new Autentificare(null,"Autentificare",
true);
```

13.4.5. JApplet

Este echivalentul din Swing pentru componenta Applet din AWT și permite realizarea de appleturi. Datorită faptului că pentru o componentă JApplet conținutul este gestionat de un panou JRootPane, apare posibilitatea de a-i adăuga acesteia bară de meniu. De asemenea, se schimbă și gestionarul de poziționare implicit care acum

va fi BorderLayout (intervine în calcul panoul conținut) în loc de FlowLayout. Nu vom insista asupra acestei componente datorită asemănării cu Applet. Pentru alte informații consultați capitolul dedicat appleturilor.

13.5. Containere intermediare

Containerele intermediare le vom numi panouri (eng. *panel*, *pane*) și sunt următoarele: JPanel, JScrollPane, JSplitPane, JTabbedPane. Panourile reprezintă modalitatea standard de a aduna mai multe componente „ușoare” la un loc și de a le gestiona poziționarea și dimensiunile folosind managerii de poziționare. Implicit, gestionarul de poziționare pentru un panou este FlowLayout. De aceea, atunci când adăugăm componente într-un panou căruia nu i-am setat alt gestionar de poziționare, acestea se vor poziționa pe aceeași linie, una după alta, cu dimensiunile lor preferate (eng. *preferred size*). Atunci când o componentă nu mai încapă pe rândul curent din cauza dimensiunilor ei, se așază pe linia următoare. Implicit, panourile nu desenează nimic altceva decât culoarea fundalului (eng. *background*) lor. Permit adăugarea de borduri și li se poate personaliza desenarea (a se vedea capitolul care descrie desenarea).

Implicit panourile sunt opace. Mai întâi vom discuta despre borduri, gestionarii de poziționare fiind deja discutați în capitolul dedicat appleturilor, în fapt fiind aceiași ca și în AWT.

13.5.1. JPanel

Clasa JPanel reprezintă containerul intermediar de bază pentru orice interfață grafică, folosit de cele mai multe ori pentru a organiza un grup de componente ale unui alt container care le conține. Se știe că și conținutul unei ferestre este ținut tot într-un panou implicit JPanel, numit contentPane, care, atenție, este gestionat implicit de gestionarul BorderLayout.

Prezentăm în continuare constructorii clasei JPanel:

Constructor	Explicație
JPanel()	Creează un panou.
JPanel(LayoutManager lm)	Creează un panou având gestionarul de poziționare lm.

Prezentăm metodele clasei JPanel care permit adăugarea, accesul și ștergerea componentelor dintr-un container, adăugarea de gestionari de poziționare, setarea bordurilor, remarcând că toate sunt moșteniri din superclasele derivate.

Metoda	Explicație
void add(Component) void add(Component, int)	Adaugă o componentă panoului. Dacă parametrul întreg este prezent, el indică indicele cu care componenta se adaugă în container.
int getComponentCount()	Returnează numărul de componente din panou.

Metoda	Explicație
Component getComponent(int i) Component getComponentAt(int x, int z) Component getComponentAt(Point p)	Permit obținerea componente cu un anumit index, având coordonatele x, y sau punctual la punctual p.
Component[] getComponents()	Returnează într-un vector toate componentele containerului.
void remove(Component c) void remove(int i) void removeAll()	Șterge componenta c, pe cea cu indicele i sau pe toate.
void setLayout(LayoutManager lm) LayoutManager getLayout()	Se setează, respectiv se obține o referință la managerul de poziționare.
public void setBorder(Border border)	Setează borderul pentru panou.

13.5.2. JScrollPane

La o primă vedere această componentă pare extrem de simplă. Scopul ei este acela de a da posibilitatea derulării (eng. *scroll*) celorlalte componente în cazul în care ele nu încap într-o zonă cu dimensiuni fixe. De notat că nu permite derularea componentelor „grele” (a se vedea capitolele introductive). De asemenea, poziționarea pentru această componentă este gestionată de un manager special numit JScrollPaneLayout și nu se pot seta gestionarii de poziționare obișnuiți. Pentru a da posibilitatea derulării unei alte componente este de ajuns să folosim un constructor al acestei clase de forma public JScrollPane(Component view), așa cum puteți vedea în secvența care urmează:



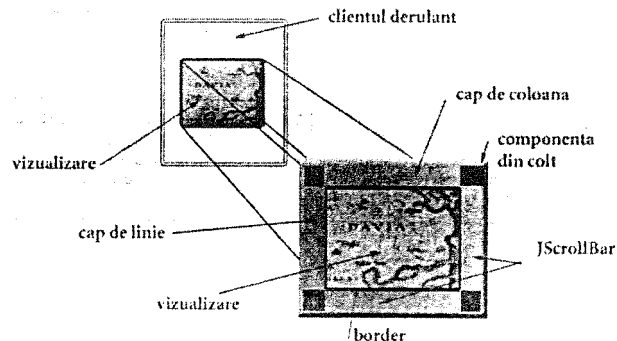
```
public JTree faArbore()
{
    //...
    DefaultMutableTreeNode radacina = new DefaultMutableTreeNode
    ("Fonduri \u015Fi colec\u0163ii");
    radacina.add(StareaCivila=newDefaultMutableTreeNode(
        "Starea civil\u0103"));
    //...
    return new JTree (radacina);
}
```

```

}
JTree arbore = faArbore();
JScrollPane derulareArbore = new JScrollPane(arbore);
this.add(derulareArbore);

```

Modul în care este proiectată această componentă este prezentat schematic în figura următoare:



Componenta `JScrollPane` oferă o perspectivă (eng. *viewport*) asupra unei surse de date care poate fi, de exemplu, o imagine, un document text, o tabelă etc. Această sursă de date reprezintă componenta din a cărei suprafață se va vizualiza doar o anumită regiune la un moment dat, cu posibilitatea de derulare. Pentru această operație `JScrollPane` pune la dispoziție două bare de derulare `JScrollBar`, respectiv un container `JViewport` care reprezintă zona prin care se vizualizează componenta conținut, care poartă denumirea de *privești* (eng. *view*) și care poate fi obținută printr-un apel `getViewport()`. Pentru a obține o referință sau pentru a seta această componentă din care doar o porțiune este vizualizată la un moment dat, putem folosi metodele `getView()` respectiv `setView()`. Dacă nu folosiți un constructor adecvat, pentru a adăuga o componentă unui `JScrollPane` folosiți un apel de forma:

```
derulareArbore.setViewportView(arbore);
```

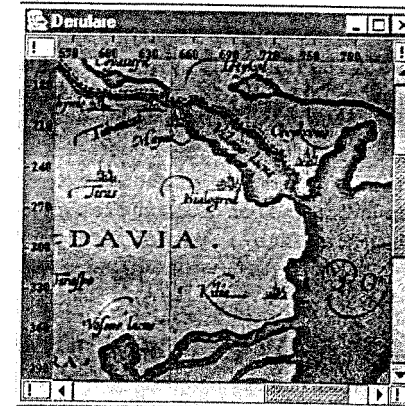
Pe lângă aceste elemente indispensabile, componenta `JScrollPane` mai pune la dispoziție un `Viewport` care are rol de cap de rând și un `Viewport` care are rol de cap de coloane. Aceste componente se vor mișca în mod sincronizat cu conținutul care se derulează. Imaginați-vă că sunt foarte utile în unele cazuri precum riglele (eng. *roolers*) în aplicațiile de desenare.

De asemenea, patru componente vor sta în colțurile rămase libere și pe care utilizatorul le poate suplini după preferință. Aceste colțuri vor fi vizibile dacă barele de derulare și capetele care le înconjoară sunt vizibile. Folosind metoda `setCorner()` pe care componenta `JScrollPane` o pune la dispoziție, putem să asociem o componentă colțului. Această metodă primește o componentă care va fi asignată și un șir de caractere care specifică colțul unde se va face acest lucru.

Alte metode pe care clasa `JScrollPane` le pune la dispoziție:

Prototipul metodei	Explicație
<code>public void setViewport (JViewport privești)</code>	Setează un nou container pentru <code>JScrollPane</code> . Va șterge vechiul <code>Viewport</code> dacă există.
<code>public void setViewportBorder (Border bordura)</code>	Setează bordura care va înconjura <code>Viewport</code> -ul componentei <code>JScrollPane</code> .
<code>public void setVertical ScrollBarPolicy(int politica)</code>	Setează modalitatea în care se va face afișarea barelor de derulare. Primește drept parametri constante după cum urmează: <code>JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED</code> va determina afișarea barelor doar când este nevoie; <code>JScrollPane.VERTICAL_SCROLLBAR_NEVER</code> nu se vor afișa niciodată barele de derulare; <code>JScrollPane.VERTICAL_SCROLLBAR_ALWAYS</code> vor fi afișate întotdeauna.
<code>public void setWheel ScrollingEnabled (boolean cu rotite)</code>	Se setează dacă sau nu se poate face scroll cu roțița mouse-ului.
<code>public void setRowHeader (JViewport capLinie)</code>	Adaugă un nou cap de linii, ștergându-l pe cel existent, dacă există. A se observa că este tot un <code>JViewport</code> .
<code>public void setColumnHeader (JViewport capColoane)</code>	Adaugă un nou cap de coloane, ștergându-l pe cel existent, dacă există.

Multe dintre aceste metode le veți regăsi în aplicația care urmează:



```

import java.awt.*;
import javax.swing.*;

public class Derulare
extends JFrame
{

```

```

private JLabel eticheta;

public Derulare() {
    super("Derulare");
    ImageIcon ii = new ImageIcon("harta.jpg");
    eticheta = new JLabel(ii);
    JScrollPane jsp = new JScrollPane(eticheta);

    JLabel[] colturi = new JLabel[4];
    for(int i=0;i<4;i++) {
        colturi[i] = new JLabel(" !");
        colturi[i].setBorder(BorderFactory.createCompoundBorder(
            BorderFactory.createEmptyBorder(1,1,1,1),
            BorderFactory.createLineBorder(Color.GRAY, 1)));
    }

    JLabel capLinii = new JLabel() {
        Font f = new Font("Serif",Font.BOLD,10);
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            Rectangle r = g.getClipBounds();
            g.setFont(f);
            g.setColor(Color.BLUE);
            for (int i = 30-(r.y % 30);i<r.height;i+=30) {
                g.drawLine(0, r.y + i, 3, r.y + i);
                g.drawString("" + (r.y + i), 6, r.y + i + 3);
            }
        }

        public Dimension getPreferredSize() {
            return new Dimension(23,(int)eticheta.
                getPreferredSize().getHeight());
        }
    };

    capLinii.setBackground(Color.GRAY);
    capLinii.setOpaque(true);

    JLabel capColoane = new JLabel() {
        Font f = new Font("Serif",Font.BOLD,10);
        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            Rectangle r = g.getClipBounds();

```

```

        g.setFont(f);
        g.setColor(Color.BLUE);
        for (int i = 30-(r.x % 30);i<r.width;i+=30) {
            g.drawLine(r.x + i, 0, r.x + i, 3);
            g.drawString("" + (r.x + i), r.x + i - 10, 16);
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension((int)eticheta.
            getPreferredSize().getWidth(),17);
    }
};

capColoane.setBackground(Color.GRAY);
capColoane.setOpaque(true);

jsp.setRowHeaderView(capLinii);
jsp.setColumnHeaderView(capColoane);
jsp.setCorner(JScrollPane.LOWER_LEFT_CORNER, colturi[0]);
jsp.setCorner(JScrollPane.LOWER_RIGHT_CORNER, colturi[1]);
jsp.setCorner(JScrollPane.UPPER_LEFT_CORNER, colturi[2]);
jsp.setCorner(JScrollPane.UPPER_RIGHT_CORNER, colturi[3]);

getContentPane().add(jsp);
setSize(300,300);
setVisible(true);
}

public static void main(String[] args) {
    new Derulare();
}
}

```

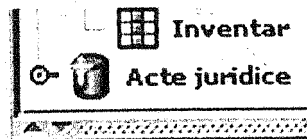
De asemenea, `JScrollPane` este destul de maleabilă încât să ofere posibilitatea realizării operației de *scroll* fără folosirea barelor de derulare și glisarea componentei conținut folosind doar mouse-ul.

13.5.3. JSplitPane

Componenta `JSplitPane` încapsulează două panouri alăturate separate printr-un marcaj despărțitor (eng. *divider*), permițând vizualizarea simultană a două componente una lângă alta (sau una deasupra celeilalte). Zona despărțitoare poate fi mutată folosind mouse-ul, mărindu-se astfel spațiul pentru unul sau altul dintre panouri, fără

a modifica suprafața totală de vizualizare. La rândul lor, panourile unui `JSplitPane` se pot divide așa cum se va vedea în exemplul alăturat. Acolo unde este cazul, o componentă se introduce într-un panou `JScrollPane`, care se va adăuga, mai apoi, unui panou al componentei `JSplitPane`, permițându-i-se derularea conținutului.

Două mici săgeți apar într-o margine a marcatului despărțitor, așa cum se poate vedea în captura care urmează. Acestea permit utilizatorului să extindă sau să închidă integral unul dintre panourile componentei `JSplitPane`. Modul în care apar este controlat de plugg-in-ul pe care aplicația îl folosește. Utilizatorul poate seta dacă ele apar sau nu folosind metoda public `void setOneTouchExpandable (boolean newValue)`.



Pentru a construi un panou despărțitor putem folosi constructorul clasei `JSplitPane`:

```
JSplitPane splitPane = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, componentaStanga,
    componenta Dreapta);
```

Primul parametru este o constantă care determină orientarea panoului, în cazul nostru pe orizontală. Dacă vrem ca cele două panouri să fie situate unul deasupra altuia putem folosi `JSplitPane.VERTICAL_SPLIT`. Orientarea poate fi schimbată și prin intermediul metodei public `void setOrientation(int orientation)`, ai cărei parametri sunt aceleași constante deja descrise.

Conținutul panourilor unui `JSplitPane` poate fi modificat dinamic folosind metodele din tabelul următor:

Metoda	Descriere
<code>public void setTopComponent (Component comp)</code>	Adaugă o componentă comp în panoul de sus.
<code>public void setLeftComponent (Component comp)</code>	Adaugă o componentă comp în panoul din stânga.
<code>public void setBottomComponent (Component comp)</code>	Adaugă o componentă comp în panoul de jos.
<code>public void setRightComponent (Component comp)</code>	Adaugă o componentă comp în panoul din dreapta.

Metodele `setTopComponent()` și `setLeftComponent()` sunt echivalente, la fel ca și metodele `setBottomComponent`, respectiv `setRightComponent`, într-un anumit fel nefiind importantă orientarea.

Panourile care compun `JSplitPane` respectă dimensiunile minime ale componentelor (eng. *minimum size*) care au fost setate pentru acestea folosind `setMinimumSize()` sau prin suprascrierea metodei `getMinimumSize()` pentru componentele personalizate. De aceea, atunci când dimensiunile panourilor sunt mai mici decât dimensiunile minime ale componentei, marcatul despărțitor nu va putea fi mutat cu mouse-ul, ci eventual panourile pot fi extinse/restrânse total folosind săgețile ajutoare. Pentru a controla deplasarea cu mouse-ul a marcatului despărțitor putem seta dimensiunea minimă a componentelor conținute. De asemenea, setând dimensiunea minimă la 0 putem restrânge total dimensiunea panourilor componentei `JSplitPane`.

Poziția marcatului despărțitor se poate schimba programatic, folosind apeluri de forma `splitPane.setDividerLocation(200)` pentru a muta la 200 de pixeli relativ la stânga/sus sau `splitPane.setDividerLocation(0.75)` la 75% relativ la stânga/sus. De asemenea, putem specifica pentru o componentă modul în care se va redistribui spațiul după ce se realizează o redimensionare a ferestrei. Acest lucru se poate rezolva printr-un apel al metodei public `void setResizeWeight(double valoare)`, unde dacă valoarea este 0 panoul din dreapta/jos va primi tot spațiul nou, iar dacă valoarea este 1, atunci panoul din stânga/sus va obține tot spațiul nou. Un exemplu de folosire a acestei componente găsiți în secțiunea care urmează, unde o folosim în combinație cu `JTabbedPane`.

13.5.4. JTabbedPane

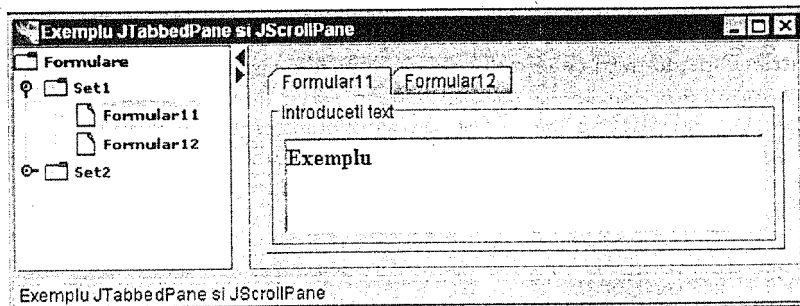
`JTabbedPane` reprezintă o stivă de componente (de obicei, panouri `JPanel` sau container, în general) așezate pe mai multe straturi suprapuse, unul dintre ele putând fi selectat la un moment dat, determinând componenta conținută să devină vizibilă. Straturile au câte o prelungire (eng. *tab*) care are proprietățile unei etichete (așa cum se poate vedea în figura alăturată, unde prezentăm o astfel de componentă având patru straturi), putând conține un icon, culoare de fundal sau tooltip.

Data/Ora: Jan 4, 1988 12:50:23 AM Progres: 0%

Interogări Statistici Actualizare SQL		
Intrări ☒ Nume ☒ Prenume ☒ Data nașterii / / ☒ Localitatea ~ ☐ Între anii -	Ieșiri ☒ Număr registru ☒ Număr act ☒ Localitatea ☐ Toate câmpurile ☐ Cota	Setări ☒ Diacritice ☒ Coduri localități ☒ Căutare rapidă
De acord Anulează		

Pentru a adăuga componente unui `JTabbedPane` putem folosi diverse supraîncărcări ale metodelor `add()`, `addTab()` sau `insertTab()` pe care clasa `JTabbedPane` le pune la dispoziție. Pentru fiecare componentă adăugată unui panou `JTabbedPane` se va crea un strat unde aceasta va fi așezată, care va avea o prelungire având forma unei etichete prin intermediul căreia utilizatorii vor selecta stratul pe care vor să-l vizualizeze. Fiecare strat are asociat un indice, numerotarea implicită pornind de la stânga spre dreapta începând cu 0. Dacă numărul de straturi este mai mare decât 0, va exista întotdeauna un indice care va fi selectat, care corespunde stratului vizibil. Implicit, este selectat stratul cu indicele 0. Indicele curent se obține folosind metoda având prototipul public `int getSelectedIndex()`.

Pentru a selecta programatic vizualizarea unui anumit strat vom folosi un apel `setSelectedIndex(int index)`.



```
// construim continut
protected void faContinut() throws Exception
{
    final JTree arbore=faArbore();

    JScrollPane panouStanga= new JScrollPane(arbore);
    panouStanga.setMinimumSize(new Dimension(150,0));

    final JScrollPane panouDreapta= new JScrollPane();
    panouDreapta.setBorder(BorderFactory.createCompoundBorder(
        new EtchedBorder (EtchedBorder.LOWERED),
        BorderFactory.createEmptyBorder(10,10,10,10)));
    panouDreapta.setPreferredSize(new Dimension(400,300));

    final JTabbedPane set1 = new JTabbedPane();

    JPanel formular11=new JPanel();
    formular11.setLayout(new BorderLayout());
    final JTextArea text = new JTextArea("");
    //text.setRows(4);
    formular11.setBorder(BorderFactory.createCompoundBorder(
```

```
BorderFactory.createTitledBorder(BorderFactory.
    createEtchedBorder(),"Introduceti text"),Border
    Factory.createEmptyBorder(5,5,5,5)));
text.setBorder(BorderFactory.createLoweredBevelBorder());
text.setWrapStyleWord(true);
text.setFont(new Font("Serif", Font.BOLD, 16));
text.setLineWrap(true);
formular11.add(text);
```

```
JPanel formular12=new JPanel(new FlowLayout());
JCheckBox cb1=new JCheckBox("Camp1");
JTextField text1=new JTextField("Camp1");
JCheckBox cb2=new JCheckBox("Camp2");
JTextField text2=new JTextField("Camp2");
```

```
formular12.add(cb1);
formular12.add(text1);
formular12.add(cb2);
formular12.add(text2);
```

```
set1.addTab( "Formular11", null, formular11 );
set1.addTab( "Formular12", null, formular12 );
```

```
final JTabbedPane set2 = new JTabbedPane();
```

```
JPanel formular21=new JPanel();
JPanel formular22=new JPanel();
set2.addTab( "Formular21", null, formular21 );
set2.addTab( "Formular22", null, formular22 );
```

```
arbore.addTreeSelectionListener(new
TreeSelectionListener() {
    public void valueChanged(TreeSelectionEvent e)
    {
        String descendent;
        DefaultMutableTreeNode nod = (DefaultMutableTreeNode)
        arbore.getLastSelectedPathComponent();
        if (nod == null) return;
        Object nodInfo = nod.getUserObject();
        String informatie=(String)nodInfo;
        DefaultMutableTreeNode parinte=(
            DefaultMutableTreeNode)nod.getParent();
        if (parinte != null)
            descendent=(String)parinte.getUserObject();
        informatie = (String)nodInfo;
```

```

System.out.println(informatie);
if (informatie=="Set1")
    panouDreapta.setViewportView(set1);
else
    if (informatie=="Set2")
        panouDreapta.setViewportView(set2);
    else
        if (informatie=="Formular11")
        { set1.setSelectedIndex(0);
          panouDreapta.setViewportView(set1);
        } else
        if (informatie=="Formular12")
        { set1.setSelectedIndex(1);
          panouDreapta.setViewportView(set1);
        } else
        if (informatie=="Formular21")
        { set2.setSelectedIndex(0);
          panouDreapta.setViewportView(set2);
        } else
        if (informatie=="Formular22")
        { set2.setSelectedIndex(1);
          panouDreapta.setViewportView(set2);
        } else
        panouDreapta.setViewportView(new JLabel());
        panouDreapta.repaint();
    });

JPanel panouJos= new JPanel();
panouJos.setLayout(new BorderLayout());
panouJos.add(faBaraStare());
panouJos.setMinimumSize(new Dimension(0,20));
panouJos.setSize(new Dimension(0,20));

JSplitPane separatorStDr=new JSplitPane(JSplitPane.
    HORIZONTAL_SPLIT, panouStanga, panouDreapta);
separatorStDr.setOneTouchExpandable(true);
JSplitPane separatorSusJos=new JSplitPane(JSplitPane.
    VERTICAL_SPLIT, separatorStDr, panouJos);
Toolkit t=this.getToolkit();

separatorSusJos.setDividerSize(5);
separatorSusJos.setResizeWeight(1);

final JPanel continut= new JPanel();
// in acest panou vom adauga continutul ferestrei

```

```

this.getContentPane().add(continut);
continut.setLayout(new BorderLayout());
continut.add(separatorSusJos);
}

public JTree faArbore()
{
    DefaultMutableTreeNode ramural1;
    DefaultMutableTreeNode ramura2;
    DefaultMutableTreeNode radacina =
        new DefaultMutableTreeNode("Formulare");

    radacina.add(ramural1=
        new DefaultMutableTreeNode("Set1") );
    radacina.add(ramura2=
        new DefaultMutableTreeNode("Set2") );
    ramural1.add( new DefaultMutableTreeNode("Formular11"));
    ramural1.add( new DefaultMutableTreeNode("Formular12"));
    ramura2.add( new DefaultMutableTreeNode("Formular21"));
    ramura2.add( new DefaultMutableTreeNode("Formular22"));
    //...
    final JTree arbore = new JTree(radacina);
    DefaultTreeCellRenderer con =
        new DefaultTreeCellRenderer();
    con.setFont(new Font("Verdana",Font.BOLD,10));
    arbore.setCellRenderer(con);
    return arbore;
}

protected JPanel faBaraStare()
{
    JPanel ajutor= new JPanel();
    // panoul care va tine eticheta de stare
    ajutor.setBorder(new SoftBevelBorder (
        SoftBevelBorder.RAISED));
    ajutor.setLayout(new BorderLayout());
    // punem in evidenta panoul prin bordura
    JLabel stare=
        new JLabel("Exemplu JTabbedPane si JScrollPane");
    // setam textul etichetei
    ajutor.add(stare,BorderLayout.CENTER);
    return ajutor;
}

```


13.6. Componente atomice simple

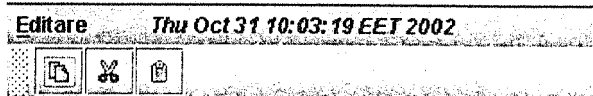
13.6.1. JLabel

Etichetele permit afișarea de informații care nu pot fi selectate sau suprascrise. Spre exemplu, se pot afișa text sau imagini, separat sau împreună, fără a se aștepta acțiuni din partea utilizatorilor, deoarece o etichetă nu poate deține comunicarea cu tastatura (eng. *focus*). Pentru interacțiune se vor folosi butoane, etichetele putând, eventual, fi folosite pentru a descrie butoane sau alte componente.

Pentru a adăuga și poziționa conținutul unei etichete se poate folosi unul dintre constructorii puși la dispoziție sau metoda `setText()` pentru a adăuga text, respectiv `setIcon()` pentru a adăuga imagini. Aliniamentul conținutului unei etichete se poate seta cu una dintre metodele `setHorizontalAlignment(int aliniament)` sau `setVerticalAlignment(int aliniament)`, unde aliniament este una din constantele definite în interfața `SwingConstants`. Implicit, textul este identat la dreapta, pe când iconurile sunt centrate. Vertical, conținutul este aliniat implicit centrat indiferent dacă este text sau imagine. Pentru o etichetă se mai pot seta culoarea de fundal, culoarea textului, fontul etc. folosind metodele pe care clasa `JLabel` le pune la dispoziție.

Un lucru interesant este faptul că o etichetă poate interpreta cod HTML afișând rezultatul obținut. Și alte componente Swing sunt capabile de acest lucru sau vor fi capabile în viitor. În cazul versiunii librăriei Swing care vine cu distribuția 1.4 a platformei Java, se poate folosi marcaj HTML și în cazul componentelor `JTabbedPane`, `JMenuItem`, `JToolTip`, `JRadioButton` și `JCheckBox`.

Prezentăm, în continuare, o clasă care suprascrie `JLabel` pentru a crea o etichetă având drept conținut un ceas a cărui prezentare este formatată folosind HTML. Eticheta este adăugată meniului bară al unei aplicații.



```
// Eticheta care afiseaza un ceas
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
```

```
public class CeasEticheta extends JLabel implements Runnable {
    private Thread ceas;
    private int reactualizare = 900;

    public CeasEticheta() {
        ceas = new Thread(this);
```

```
        ceas.start();
    }

    public void reactualizeaza() {
        Date data=new Date();
        String dataNeFormatata=data.toString();
        String dataFormatata="<html> <font color=blue> <i>"+
            dataNeFormatata+"</i> </font> </html>";
        setText(dataFormatata);
    }

    // implementarea interfetei Runnable
    public void run() {
        try {
            while (true) {
                reactualizeaza();
                Thread.sleep(reactualizare);
            }
        } catch (InterruptedException e) {
        }
    }
}
```

Adăugarea etichetei meniului:

```
menuBar.add(editare);
// adaugam meniul optiuni la bara de meniu
menuBar.add(new JLabel("      "));
menuBar.add(new CeasEticheta());
setJMenuBar(menuBar);
```

Etichetele pot fi folosite și drept mijloc de a transfera focusul spre alte componente folosind metoda `setLabelFor()` din clasa `JLabel`. În acest sens prezentăm o metodă care setează conținutul unei ferestre:



```
protected void faContinut() throws Exception
{
    JPanel p1 = new JPanel();
    JLabel et1 = new JLabel("Utilizator:"); // apel constructor
    et1.setDisplayedMnemonic('U');
    p1.add(et1, BorderLayout.EAST);
    JTextField tel = new JTextField(7);
```

```

tel.setToolTipText("Introduceti utilizator");
et1.setLabelFor(tel);
p1.add(tel, BorderLayout.WEST);

JPanel p2 = new JPanel();
JLabel et2 = new JLabel("Parola:");
et2.setDisplayedMnemonic('P');
p2.add(et2, BorderLayout.EAST);
JTextField te2 = new JTextField(7);
te2.setToolTipText("Introduceti parola");
et2.setLabelFor(te2); // transfer focus folosind eticheta
p2.add(te2, BorderLayout.WEST);

JPanel continut=new JPanel(new GridLayout(0,2));
continut.add(p1);
continut.add(p2);

this.getContentPane().add(continut, BorderLayout.CENTER);
}

```

13.6.2. Butoane

Toate componentele reprezentând butoane în Swing extind clasa `AbstractButton`. De remarcat că și clasele `JMenuItem`, `JCheckBoxMenuItem` și `JRadioButtonMenuItem` care desemnează opțiuni într-un meniu extind `AbstractButton`. Prezentăm în tabelul care urmează o listă a componentelor grafice având rol de butoane, cu un exemplu de reprezentare grafică, urmat de explicațiile corespunzătoare:

Buton	Vizualizare	Explicație
JButton		Un buton simplu a cărui apăsare determină aruncarea unui eveniment care trebuie tratat de programator. Prin apăsare, butonul își schimbă și vizualizarea, iar după eliberarea butonului mouse-ului se va reveni la forma inițială.
JToggleButton		Un buton care permite comutarea între două stări. Permite personalizări ale butoanelor <code>JCheckBox</code> și <code>JRadioButton</code> , fiind și superclasă pentru acestea.
JCheckBox	☐ Nume ☒ Nume	Un buton a cărui stare poate fi setată între bifat sau nebifat.
JRadioButton	☐ Mod grafic ☒ Mod grafic	Un buton radio care în general formează împreună cu alte butoane radio un grup din care numai unul poate fi apăsător la un moment dat.

Stările în care se poate găsi un buton la un moment dat sunt date de următoarele proprietăți:

Stare	Explicații
Selectat (eng. <i>selected</i>)	Arată dacă s-a schimbat starea butonului prin apăsarea și apoi eliberarea butonului mouse-ului (această proprietate are valoare doar în cazul <code>JToggleButton</code> care are două stări bifat/nebifat).
Apăsător (eng. <i>pressed</i>)	Are valoarea <code>true</code> atunci când butonul mouse-ului a fost eliberat, după ce inițial a fost apăsător.
Derulant (eng. <i>rollover</i>)	Are valoarea <code>true</code> atunci când se tratează cazul în care cursorul mouse-ului se află deasupra suprafeței butonului.
Armat (eng. <i>armed</i>)	Un buton se află în această stare dacă este apăsător cu mouse-ul fără a ridica degetul de pe buton; este folosită pentru a opri aruncarea evenimentului care generează tratarea, atunci când am apăsător un buton și mai apoi îl eliberăm având cursorul mouse-ului în afara suprafeței butonului (renunțăm să mai apăsăm butonul).
Activat (eng. <i>enabled</i>)	Are valoarea <code>true</code> când butonul poate recepta acțiunile care se fac asupra lui cu mouse-ul sau tastatura (deține focusul) și în acest caz celelalte proprietăți nu pot să-și schimbe starea când e <code>false</code> .

Dacă apăsăm cu mouse-ul un buton simplu, `JButton`, acesta se va „arma”, pe când pentru un `JToggleButton` acesta va deveni selectat.

Butoanele folosesc un model pentru a gestiona stările în care se pot afla la un moment dat. Acesta poartă numele `ButtonModel` și reprezintă baza pentru realizarea de modele pentru butoane. Putem avea acces la acest model folosind metodele `getModel()`, respectiv `setModel()`, prima pentru a obține modelul, iar a doua pentru a seta un nou model pentru butonul curent. Dacă nu precizăm nici un model, implicit se va asocia `DefaultButtonModel`, care reprezintă implementarea implicită care ne este pusă la dispoziție pentru modelul `ButtonModel`. În general, în practică, nu este nevoie să creăm propriul model pentru butoane, fiind îndeajuns să folosim unul dintre constructorii puși la dispoziție.

13.6.2.1. JButton

Un buton poate prezenta atât text, cât și imagini, toate aspectele referitoare la acest subiect fiind comune cu cele discutate în cazul etichetelor. O imagine poate fi atașată unui buton folosind constructorul care are ca parametru o instanță a clasei `Icon` sau după desenare, folosind metoda `setIcon(Icon imagine)`. Putem adăuga iconuri care să apară în diferite stări ale butonului, când este selectat, apăsător, dezactivat etc.

Unui buton i se poate asocia drept accelerator o literă din textul pe care îl prezintă drept conținut (aceasta va deveni subliniată) folosind un apel de genul `setMnemonic('B')`. Apăsarea tastei `ALT` și a tastei reprezentând litera respectivă determină declanșarea acțiunii.

Tratarea evenimentelor generate de butoane se face în funcție de tipul de buton. În general, se adaugă ascultători `ActionListener` care tratează evenimentul aruncat de apăsarea butonului. În cazul `JCheckBox` și `JRadioButton` se folosește un ascultător

ItemListener pentru a testa dacă butonul este sau nu selectat. Pentru exemple de implementare a ascultătorilor puteți consulta subcapitolul referitor la evenimente și ascultători.

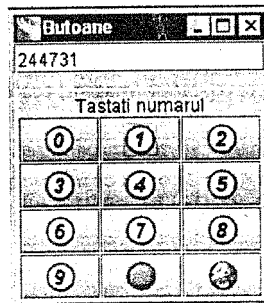
Un singur buton al interfeței grafice poate deține legătura cu tastatura la un moment dat, ceea ce înseamnă că apăsarea tastei spațiu (eng. *space-bar key*) în acel moment determină declanșarea acțiunii asociate butonului. Pentru a selecta butonul implicit care deține focusul la startarea aplicației, folosim un apel `fereastrăMea.getRootPane().setDefaultButton(buton)`. Un buton poate fi dezactivat (nu va mai răspunde la acțiunile utilizatorilor efectuate asupra lui) folosind metoda `setEnabled(boolean stare)`.

13.6.2.2. JToggleButton, JCheckBox, JRadioButton

Componentele `JCheckBox`, `JRadioButton` și `JToggleButton` au aceeași funcționalitate, permițând simularea unui comutator (eng. *switch*) cu două stări. Singura diferență dintre toate aceste componente o reprezintă delegatii-UI care le dau vizualizarea și care sunt diferiți. De remarcat faptul că `JCheckBox` și `JRadioButton` moștenesc clasa `JToggleButton`, care oferă pentru acestea mecanismul de comutare între cele două stări posibile. Putem verifica dacă un buton este selectat (spre exemplu, pentru un buton `JCheckBox` acest lucru înseamnă că e bifat) folosind metoda `isSelected()` și îl putem selecta efectiv folosind metoda `setSelected(boolean stare)`.

Folosind clasa `ButtonGroup` putem grupa mai multe butoane împreună pentru a garanta că numai unul poate fi selectat la un moment dat. Gruparea se poate face numai pentru butoanele care moștenesc `JToggleButton`, deoarece numai acestea sunt selectabile.

Exemple de folosire a butoanelor găsiți în mai toate aplicațiile acestui capitol. Vom prezenta câteva metode în care sunt cuprinse multe dintre aspectele discutate în această secțiune. Acestea permit atașarea drept conținut pentru o fereastră `JFrame` a unui panou pentru introducerea numerelor.



```
// construim continut
protected void faContinut() throws Exception
{
    JPanel panouSus= new JPanel();
```

```
panouSus.setLayout(new BorderLayout());
ecran=new JTextField();
panouSus.add(ecran);
panouSus.setMinimumSize(new Dimension(500,20));

JSplitPane separatorSusJos=new JSplitPane(
    JSplitPane.VERTICAL_SPLIT, panouSus, faPanouJos());

separatorSusJos.setResizeWeight(0);

final JPanel continut= new JPanel();
// in acest panou vom adauga continutul ferestrei
this.getContentPane().add(continut);
continut.setLayout(new BorderLayout());
continut.add(separatorSusJos);
}

protected JPanel faPanouJos() // construiesc panou butoane
{
    JPanel panou= new JPanel();
    panou.setBorder(new
        SoftBevelBorder(SoftBevelBorder.RAISED));
    panou.setLayout(new BorderLayout());
    JLabel mesaj=new JLabel("Tastati numarul",
        SwingConstants.CENTER);
    panou.add(mesaj, BorderLayout.NORTH);

    JPanel tastatura=new JPanel();
    tastatura.setLayout(new GridLayout(0,3));
    Ascultator asc= new Ascultator() // definire ascultator
    {
        public void actionPerformed(ActionEvent e)
        {
            String sir=ecran.getText();
            String nou=e.getActionCommand();
            int nr=new Integer(nou).intValue();
            if (nr<10)
                ekran.setText(sir+nou); // introducere numar in zona editare
            else
                if (nr==10)
                    ekran.setText(""); // reseteaza text
                else
                    iesire(); // parasirea ferestrei curente
        }
    };
};
```

```

for(int i=0;i<12;i++)
{
    final int aux=i;
    JButton tasta= new JButton(
        new ImageIcon("C"+aux+".gif"))
    {
        public String getActionCommand()
        {
            return new Integer(aux).toString();
        }
    };
    tastatura.add(tasta); // adaugam buton panoului
    tasta.addActionListener(asc); // atasam ascultator butonului
}

panou.add(tastatura);
return panou;
}

public void iesire() {
    System.exit(0); // parasim aplicatia
}

```

13.6.3. Borduri

Rolul bordurilor (eng. *borders*) este important în realizarea unei interfețe grafice prietenoase. Prin intermediul lor putem crea butoane care să se distingă de context, umbre pentru zonele de text, putem separa între ele componentele. Bordurile nu sunt componente (nu moștenesc *JComponent*), astfel încât nu le putem asocia ascultători, tooltips etc. Clasa din care sunt derivate toate bordurile, și anume *AbstractBorder*, este derivată direct din clasa *Object*, la fel ca toate clasele din Java. Clasele reprezentând diverse borduri se află localizate în pachetul *javax.swing* care trebuie importat pentru a le putea folosi.

Swing pune la dispoziția utilizatorilor o serie de borduri standard care pot fi folosite în obținerea de borduri compuse, personalizate. Lista lor o prezentăm în continuare:

Bordura	Explicația
<i>CompoundBorder</i>	O combinație de două borduri: una interioară și una exterioară. Reprezintă metoda prin care se pot compune borduri.
<i>EmptyBorder</i>	O bordură transparentă folosită doar pentru a defini spațiul liber în jurul componentelor.
<i>EtchedBorder</i>	O bordură având forma unei linii care apare în relief.
<i>LineBorder</i>	O bordură plană cu o grosime și culoare specificate.
<i>MatteBorder</i>	O bordură constând dintr-o culoare sau o mică imagine.

Bordura	Explicația
<i>SoftBevelBorder</i>	O bordură 3D ridicată sau coborâtă, cu colțurile rotunjite.
<i>TitledBorder</i>	O bordură care afișează un titlu într-o anumită poziție. Putem specifica fontul titlului, culoarea, identarea și poziționarea folosind metodele și constantele clasei <i>TitleBorder</i> .

Pentru a seta bordura unei componente se utilizează metoda *setBorder()* pe care acestea o moștenesc din clasa *JComponent*. Mai există, de asemenea, o clasă care poartă denumirea de *BorderFactory*, care se află în pachetul *javax.swing*, care conține metode statice ce pot fi folosite pentru a construi borduri de diferite tipuri. Spre exemplu, secvența următoare atașează o bordură goală panoului panou, distanțându-l astfel cu 10 pixeli de celelalte componente care sunt așezate lângă el.

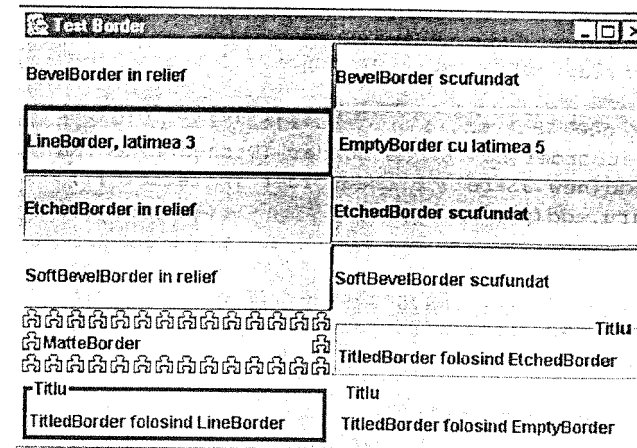
```

import javax.swing.*;
//...
JPanel panou= new JPanel();
panou.setBorder(BorderFactory.createEmptyBorder(
    10, //sus
    10, //stanga
    10, //jos
    10) //dreapta
);

```

În general, clasele reprezentând borduri nu conțin metode pentru modificarea dimensiunilor, culorilor etc. Pentru aceasta trebuie creată o nouă instanță a bordurii care este asociată componentelor. De asemenea, se poate atașa o aceeași bordură mai multor componente.

Prezentăm în continuare o aplicație care ilustrează câteva tipuri de borduri pe care le vom adăuga unor butoane (același lucru se poate face și pentru panouri, metoda *setBorder()* fiind moștenită pentru toate componentele din clasa *JComponent*):



```

import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

class TestBorder extends JFrame
{
    public TestBorder() {
        setTitle("Test Border");
        setSize(200, 400);

        JPanel cadru = (JPanel) getContentPane();
        cadru.setLayout(new GridLayout(6,2));

        JButton p = new JButton();
        p.setBorder(new BevelBorder (BevelBorder.RAISED));
        p.add(new JLabel("BevelBorder in relief"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new BevelBorder (BevelBorder.LOWERED));
        p.add(new JLabel("BevelBorder scufundat"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new LineBorder (Color.blue, 3));
        p.add(new JLabel("LineBorder, latimea 3"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new EmptyBorder (5,5,5,5));
        p.add(new JLabel("EmptyBorder cu latimea 5"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new EtchedBorder (EtchedBorder.RAISED));
        p.add(new JLabel("EtchedBorder in relief"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new EtchedBorder (EtchedBorder.LOWERED));
        p.add(new JLabel("EtchedBorder scufundat"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new SoftBevelBorder (SoftBevelBorder.RAISED));

```

```

        p.add(new JLabel("SoftBevelBorder in relief"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new SoftBevelBorder
            (SoftBevelBorder.LOWERED));
        p.add(new JLabel("SoftBevelBorder scufundat"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new MatteBorder (
            new ImageIcon("imprejur.gif")));
        p.add(new JLabel("MatteBorder"));
        cadru.add(p);

        p = new JButton();
        TitledBorder b=new TitledBorder(new EtchedBorder(2,
            "Titlu"));
        b.setTitleJustification(TitledBorder.RIGHT);
        p.setBorder(b);
        p.add(new JLabel("TitledBorder folosind EtchedBorder"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new TitledBorder ( new LineBorder (Color.RED,
            3), "Titlu"));
        p.add(new JLabel("TitledBorder folosind LineBorder"));
        cadru.add(p);

        p = new JButton();
        p.setBorder(new TitledBorder ( new EmptyBorder (5,5,5,5),
            "Titlu"));
        p.add(new JLabel("TitledBorder folosind EmptyBorder"));
        cadru.add(p);

        setVisible(true);
    }

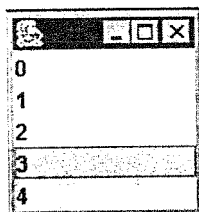
    public static void main(String args[]) {
        new TestBorder();
    }
}

```

13.6.4. JList

Această componentă permite utilizatorilor să aleagă un element dintr-o listă de obiecte. Conținutul unei liste este gestionat printr-un model care este o instanță a clasei `ListModel`. Putem preciza că `JList` oferă acces la conținutul listei doar prin intermediul modelului de date asociat. Constructorii acestei componente permit crearea de liste direct dintr-un tablou având drept elemente instanțe de clase derivate din clasa `Object` sau dintr-un vector `Vector` (care, la rândul lui, este o colecție de elemente `Object`). Ceea ce va fi afișat ca text în listă, ținând cont că elementele listei sunt instanțe `Object`, va fi șirul rezultat din apelul `toString()` al obiectelor (numai în cazul instanțelor clasei `Icon` se va face tipărirea lor ca iconuri în etichete). În cazul în care nu asociem un model de date explicit, conținutul este gestionat tot prin intermediul unui model `ListModel` implicit, și anume o instanță a `DefaultListModel`, care este alocat componentei. Clasa `DefaultListModel` este o implementare implicită a modelului `ListModel` și ține interior lista sub forma unui vector de obiecte `Object`. Mai există o implementare abstractă pentru `ListModel`, și anume `AbstractListModel`, care permite programatorului să aleagă structura de date în care să păstreze lista.

Un exemplu de listă creată folosind un tablou regăsiți în capitolul în care se enumeră componentele. Prezintă un exemplu în care construim o listă dintr-un vector.



```
Vector v=new Vector();
for(int i=0;i<5;i++)
{
    v.add(i,new Integer(i).toString());
}

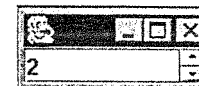
JList lista = new JList(v);
```

Aceeași vizualizare o obținem prin crearea unei liste având următorul model de date:

```
ListModel model = new AbstractListModel() {
    // este indeajuns implementarea acestor 2 metode pentru a
    // crea un model AbstractListModel
    public int getSize() { return 5; }
    public Object getElementAt(int index) { return new
    Integer(index).toString(); }
```

```
};
JList lista = new JList(model);
```

Pentru a putea derula conținutul unei liste trebuie, mai întâi, să o includem într-un `JScrollPane` printr-un apel de forma `JScrollPane derulare = new JScrollPane(lista);`



Metodele cele mai importante pe care le putem folosi în cazul unei liste sunt enumerate în următorul tabel:

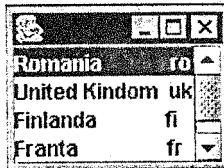
Metoda	Explicații
<code>public Object getSelectedValue()</code>	Returnează primul element selectat din lista de elemente selectate; returnează null dacă nu este nici un element selectat.
<code>public void setSelectedValue (Object unObiect, boolean deruleaza)</code>	Selectează obiectul <code>unObiect</code> specificat din listă; dacă <code>deruleaza</code> este true atunci se va derula lista pentru a se vizualiza obiectul selectat.
<code>setSelectedIndex(int index)</code>	Selectează elementul având indicele <code>index</code> .
<code>public void clearSelection()</code>	Șterge selecția curentă.
<code>public void setVisibleRowCount (int numarCeluleVizibile)</code>	În cazul în care lista este așezată într-un <code>JScrollPane</code> , această metodă asigură că numai <code>numarCeluleVizibile</code> celule vor fi direct vizibile.
<code>public void ensureIndex IsVisible (int index)</code>	În cazul în care lista este așezată într-un <code>JScrollPane</code> , apelând această metodă, lista se va derula pentru a face vizibil elementul având indicele <code>index</code> .
<code>public void setSelectionMode (int modSelectie)</code>	Setează modul de selecție; <code>modSelectie</code> poate fi una din constantele: <code>ListSelectionModel.SINGLE_SELECTION</code> - pentru a putea selecta un singur element. <code>ListSelectionModel.SINGLE_INTERVAL_SELECTION</code> - pentru a putea selecta un singur interval. <code>ListSelectionModel.MULTIPLE_INTERVAL_SELECTION</code> - pentru a putea selecta mai multe intervale.
<code>public ListModel getModel() public void setModel (ListModel model)</code>	Returnează/setează modelul care ține datele listei.

Evenimentul pe care îl poate genera o componentă `JList` este `ListSelectionEvent`, care se găsește în pachetul `javax.swing.event` și care este aruncat atunci când se selectează unul din elementele listei. Prezintă un exemplu de tratare a

acestui eveniment folosind o clasă interioară anonimă. Se va afișa în consolă indicele elementului selectat din listă:

```
lista.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent e)
    {
        int i=lista.getSelectedIndex();
        System.out.println(i);
    }
});
```

Vom prezenta în continuare un exemplu de utilizare a listelor în care utilizatorul determină prin apăsarea unei taste selectarea elementului al cărui nume începe cu litera corespunzătoare tastei. Vom folosi, de asemenea, și clasa `DesenareCelule` care implementează `ListCellRenderer` pentru a formata parțial lista (se schimbă culoarea selecției în roșu):



```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Fereastra extends JFrame {
    // constructor
    public Fereastra () throws Exception {
        String [] state = {
            "Romania      ro",
            "United Kindom uk",
            "Finlanda      fi",
            "Franta        fr",
            "Grecia        gr"};

        final JList lista = new JList(state);

        class CautaInLista extends KeyAdapter
        {
            JList lista;
            ListModel model;
            String m_key;
```

```
public CautaInLista(JList lista) {
    lista = lista;
    model = lista.getModel();
}

public void keyTyped(KeyEvent e) {
    char ch = e.getKeyChar();
    if (!Character.isLetterOrDigit(ch))
        return;

    m_key += Character.toUpperCase(ch);

    for (int k=0; k<model.getSize(); k++) {
        String str = ((String)model.getElementAt(k)).
            toUpperCase();
        if (str.startsWith(m_key)){
            lista.setSelectedIndex(k);
            lista.ensureIndexIsVisible(k);
            break;
        }
    }
}

lista.addKeyListener(new CautaInLista(lista));

class DesenareCelule extends JLabel implements
ListCellRenderer {
    public DesenareCelule() {
        setOpaque(true);

        public Component getListCellRendererComponent( JList
            list, Object value, int index, boolean isSelected,
            boolean cellHasFocus)
        {
            setText(value.toString());
            setBackground(isSelected ? Color.red : Color.white);
            setForeground(isSelected ? Color.white : Color.black);
            return this; // returneaza chiar aceasta eticheta
        }
    }

    DesenareCelule desen = new DesenareCelule();
    lista.setCellRenderer(desen);
```

```

lista.setVisibleRowCount(4);
JScrollPane derulare = new JScrollPane(lista);
this.getContentPane().add(derulare);
}
}

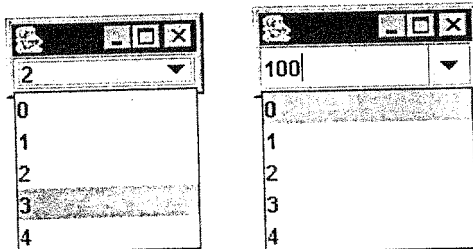
```

13.6.5. JComboBox

Componenta `JComboBox` combină un buton (care se transformă într-un câmp text când componenta este editabilă) cu o listă expandabilă la cerere din care utilizatorul poate să selecteze o singură opțiune la un moment dat. Deoarece componenta `JComboBox` are aceeași funcționalitate ca `JList`, este dificilă alegerea între aceste două posibilități. Se recomandă folosirea componentei `JComboBox` în defavoarea `JList` în cazul în care se dorește ca utilizatorul să poată să aleagă o singură opțiune pe care să o vadă explicit mai apoi. Dacă componenta este setată să fie editabilă, atunci va conține un câmp text (și nu un buton) în care utilizatorul poate introduce date de intrare. Intern, această componentă este construită dintr-un buton și un meniu `JPopupMenu` care conține o listă `JList` (reprezentând opțiunile posibile) așezată mai întâi într-un panou `JScrollPane` pentru a se putea derula.

Această componentă permite folosirea mai multor constructori asemănători componentei `JList`. Se poate construi un `JComboBox` folosind un model dat explicit care este o instanță a clasei `ComboBoxModel` și, în acest caz, vom avea acces maxim asupra proprietăților și comportamentului componentei. Putem să creăm o componentă `JComboBox` direct dintr-un tablou de obiecte `Object` sau dintr-un vector `Vector`. Mai putem crea un `JComboBox` vid, căruia îi putem asocia mai apoi un model de date. Un exemplu de `JComboBox` creat dintr-un tablou de obiecte găsiți în secțiunea destinată desenării textului. Trebuie să precizăm că avem la dispoziție clasa `DefaultComboBoxModel`, care este implementarea implicită pentru `ComboBoxModel` și pe care o putem folosi drept model efectiv pentru componentele noastre. Mai există un model, și anume interfața `MutableComboBoxModel`, ce face posibile operațiile de adăugare, ștergere și modificare în mod dinamic a conținutului componentei `JComboBox` care o folosește ca model de date.

Prezentăm un exemplu de componentă `JComboBox` construită folosind un model de date care-i gestionează conținutul (același exemplu de la listă modificat pentru `JComponent` – ne putem imagina astfel `JComponent` ca o listă derulantă):



```

import javax.swing.*;
import java.awt.*;
import java.util.*;

public class Fereastra extends JFrame {
    // constructor
    public Fereastra() throws Exception {
        DefaultComboBoxModel model = new DefaultComboBoxModel() {
            public int getSize() { return 5; }
            public Object getElementAt(int index) { return new
                Integer(index).toString(); }
        };

        JComboBox cb = new JComboBox(model);
        /* cb.setEditable(true);
        face diferenta intre cele doua captori editabila/needitabila */
        this.getContentPane().add(cb);
    }
}

```

Sintetizăm în următorul tabel principalele metode pe care clasa `JComboBox` le pune la dispoziția dezvoltatorilor, cu remarca că toate metodele care se referă la adăugare, ștergere și modificare de elemente necesită ca modelul componentei `JComboBox` curente să fie o implementare a clasei `MutableComboBoxModel`:

Metoda	Explicație
<code>public void setEnabled(boolean b)</code>	Setăm dacă componenta <code>JComboBox</code> curentă va da posibilitatea editării.
<code>void addItem(Object o)</code> <code>void insertItemAt(Object o, int poz)</code>	Se adaugă obiectul <code>o</code> în lista derulantă, respectiv se inserează înainte de obiectul de pe poziția <code>poz</code> .
<code>Object getSelectedItem()</code>	Returnează obiectul curent selectat.
<code>void removeAllItems()</code> <code>void removeItemAt(int poz)</code> <code>void removeItem(Object ob)</code>	Se șterg din lista derulantă toate obiectele, cel de pe poziția <code>poz</code> , respectiv obiectul <code>ob</code> .
<code>int getItemCount()</code>	Returnează numărul de elemente al meniului derulant.
<code>ComboBoxModel getModel()</code> <code>void setModel(ComboBoxModel)</code>	Se obține/setează modelul de date pentru componenta <code>JComboBox</code> curentă.

O componentă `JComboBox` poate genera două tipuri de evenimente, și anume: `ItemEvents` și `ActionEvents`. Prezentăm un ascultător care afișează la consolă indexul elementului selectat.

```

cb.addItemListener( new ItemListener()
{
    public void itemStateChanged(ItemEvent e)

```



```

    {
        System.out.println("Ati ales " + e.getItem().toString());
    }
});

```

Modul în care este vizualizată o componentă JComboBox este determinat de o instanță a clasei ListCellRenderer pentru lista derulantă, respectiv ComboBox Editor pentru câmpul text. Dacă nu este specificat un alt obiect care să se ocupe de modul de desenare a componentei, ceea ce se va vedea în lista derulantă va fi ceea ce generează toString() pentru obiectele componente (ca și la liste, exceptând obiectele instanțe ale clasei Icon care vor fi afișate chiar ca iconuri). Prezentăm, în continuare, un exemplu de personalizare a reprezentării grafice pentru o componentă JComboBox împreună cu un exemplu efectiv de folosire a JComboBox. Este vorba de exemplul din subcapitolul JList remodelat cu JComboBox.



```

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Fereastra3 extends JFrame {
    // constructor
    public Fereastra3 () throws Exception {
        String [] state = {
            "Romania", "ro",
            "United Kindom", "uk",
            "Finlanda", "fi",
            "Franta", "fr",
            "Grecia", "gr");

        final JComboBox cb = new JComboBox(state);

        class DesenareCelule extends JLabel implements
        ListCellRenderer {

```

```

    public DesenareCelule() {
        setOpaque(true);
    }

    public Component getListCellRendererComponent( JList
    list, Object value, int index, boolean isSelected,
    boolean cellHasFocus)
    {
        setText(value.toString());
        setBackground(isSelected ? Color.red : Color.white);
        setForeground(isSelected ? Color.white : Color.black);
        return this; // returneaza chiar aceasta eticheta
    }
}

DesenareCelule desen = new DesenareCelule();
cb.setRenderer(desen);
this.getContentPane().setLayout(new FlowLayout());
this.getContentPane().add(cb);
}

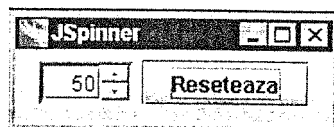
```

13.6.6. JSpinner

Reprezintă o singură linie de text care permite utilizatorului să selecteze o valoare dintr-o secvență de valori. JSpinner oferă, de obicei, o pereche de săgeți care permite navigarea prin valorile secvenței. Același lucru se poate realiza prin intermediul săgeților sus/jos. Utilizatorul poate să introducă o valoare direct în câmpul text. Componenta JComboBox oferă aceleași funcționalități, cu diferența că determină apariția unei liste derulante care prezintă dezavantajul de a se suprapune peste datele importante.

Valoarea zonei de text a componentei JSpinner se poate modifica printr-un apel setValue(Object obiect) și se poate obține printr-un apel getValue(). Componentele JSpinner pot arunca evenimente ChangeEvent atunci când datele din model se modifică, acestea putând fi captate și tratate de către ascultătorii ChangeEventListener atașați componentelor folosind metoda addChangeListener(ChangeListener ascultator).

Modelul de date pentru această componentă este o clasă care implementează SpinnerModel. Există deja trei implementări ale acestei interfețe pe care programatorii le au la dispoziție, și anume: SpinnerListModel pentru un tablou de elemente, SpinnerNumberModel pentru o listă de numere și SpinnerDateModel pentru data calendaristică (pentru dată vom folosi java.util.Calendar). Prezentăm în continuare o componentă JSpinner construită pe baza unui model SpinnerNumberModel.



```
protected void faContinut() throws Exception
{
    JPanel continut=(JPanel)this.getContentPane();
    continut.setLayout(new FlowLayout());
    Integer valoare = new Integer(50); // valoarea implicita
    Integer minim = new Integer(0); // valoarea minima
    Integer maxim = new Integer(100); // valoarea maxima
    Integer pas = new Integer(1); // pasul de deplasare
    SpinnerNumberModel model = new SpinnerNumberModel(valoare,
    minim, maxim, pas); // apel constructor model
    final JSpinner spin=new JSpinner(model);
    continut.add(spin, BorderLayout.CENTER);
    JButton but=new JButton("Reseteaza");
    continut.add(but, BorderLayout.EAST);
    but.addActionListener(new ActionListener() {
        public void actionPerformed
        (ActionEvent e) {
            spin.setValue(new Integer(50)); // resetare valoare
        }
    });
}
```

13.7. Componente atomice complexe

13.7.1. Componente text

Clasa abstractă `JTextComponent` stă la baza tuturor componentelor text din Swing și este localizată în pachetul `javax.swing.text`. Toate componentele text se regăsesc în pachetul `javax.swing` și sunt enumerate în continuare: `JTextField`, `JPasswordField`, `JTextArea`, `JEditorPane`, `JTextPane`. `JTextComponent` moștenește direct clasa `JComponent`. Conținutul unei componente text este menținut într-o instanță a interfeței `Document`, care reprezintă modelul de date pentru aceasta. Sunt oferite două implementări efective ale interfeței `Document`: `PlainDocument` și `StyledDocument`. Prima oferă un singur font, o singură culoare pentru text și conținutul format numai din caractere. Cea de a doua este mult mai complexă, permițând fonturi și culori multiple, imagini și componente scufundate în conținut. `JTextField`, `JPasswordField` și `JTextArea` au drept model `PlainDocument`, iar `JEditorPane`, `JTextPane` folosesc ca model `StyledDocument`. Putem obține sau seta modelul unei componente text folosind metodele `getDocument()`, respectiv `setDocument()`.

Fiecare componentă text are asociat un cursor (eng. *caret*) care marchează poziția curentă unde se scrie și care reprezintă o instanță a clasei `Caret`. Metodele pe care le avem la dispoziție pentru a gestiona cursorul le prezentăm în tabelul care urmează:

Metoda	Explicație
<code>setCaret(Caret c)</code> <code>getCaret()</code>	Permite setarea/obținerea cursorului asociat componentei text.
<code>setCaretColor(Color c)</code> <code>getCaretColor()</code>	Permite setarea/obținerea culorii cursorului.
<code>setCaretPosition(int pozitie)</code> <code>getCaretPosition()</code>	Permite setarea/obținerea poziției cursorului în interiorul textului.

Componentele text mențin informații despre zonele de text selectate la momentul curent. Putem obține textul selectat ca `String` folosind `getSelectedText()` și putem obține sau seta culoarea de fundal pentru zona selectată folosind metodele `getSelectionBackground()`, respectiv `setSelectionBackground()`. De asemenea, sistemul de copiere-lipire (eng. *copy/paste*) ne este accesibil folosind metodele puse la dispoziție de clasa `JTextComponent`. Programatorii le pot folosi pentru a crea meniuri pop-up sau meniuri pentru editare. Descriem în continuare în următorul tabel metodele care sunt puse la dispoziția utilizatorilor în acest scop:

Metoda	Explicație
<code>copy()</code>	Copie zona de text selectată în clipboard-ul sistemului de operare folosit.
<code>cut()</code>	Mută zona de text selectată în clipboard-ul sistemului de operare folosit.
<code>paste()</code>	Lipește la poziția curentă a cursorului ceea ce se găsește în clipboard la momentul curent.
<code>select(int pozIn, int pozFin)</code>	Selectează textul componentei text începând cu poziția de început <code>pozIn</code> și terminând cu poziția de sfârșit <code>pozFin</code> .
<code>selectAll()</code>	Selectează întreg textul componentei.

Un exemplu de folosire a acestor metode îl veți găsi în subcapitolul referitor la meniuri pop-up. Folosind combinațiile de taste (acestea sunt dependente de look-and-feel-ul folosit) `Ctrl+C`, respectiv `Ctrl+V` putem realiza *copy/paste* fără a scrie cod pentru aceasta. De asemenea, putem selecta text folosind *Shift* și tastele cursor.

Sunt puse la dispoziția programatorilor diverse metode prin care se permite manevrarea conținutului zonei de text. Astfel, conținutul unei componente text se poate seta sau obține folosind metoda `setText()`, respectiv `getText()`. Conținutul se poate, de asemenea, trimite spre un flux de ieșire folosind metoda `write(Writer iesire)` sau se poate folosi metoda `read(Reader intrare, Object descriere)` pentru a crea un model `Document` nou al cărui conținut este dat de intrare și care se va atașa componentei text, vechiul model fiind pierdut.

Prezentăm, în continuare, un exemplu care ilustrează cele trei componente text simple, și anume `JTextField`, `JTextArea` și `JPasswordField`. Vom pune în

evidență modul în care modelul de date ne ajută în crearea de componente personalizate. Extinzând modelele care stau la baza componentelor text, putem crea zone de text care să permită doar scrierea literelor sau numai a caracterelor, zone de text având un număr fix de caractere etc. Panoul care urmează reprezintă un formular care va trebui completat de către utilizatorii aplicației, conținând:

- o zonă text `CampNumar` care extinde `JTextField` oferind posibilitatea introducerii de text conținând numai cifre și având o dimensiune precizată.
- un câmp `JTextField` obișnuit care nu are nici o constrângere impusă.
- un câmp `JTextFieldPersonalizat` care permite doar introducerea unui număr precizat de caractere, care trebuie să fie litere sau spațiu.
- un câmp `JPasswordField` care reprezintă o parolă ce se așteaptă a fi introdusă.
- un câmp `Continut` care extinde `JTextArea` oferind un număr constant de caractere și determinând pierderea focusului componente la apăsarea tastei `Tab`.

Butonul `De acord` determină realizarea unei anumite acțiuni folosind conținutul colectat din zonele de text, folosind metoda `getText()`, iar butonul `Anuleaza` determină ștergerea conținutului tuturor componentelor text folosind apeluri de genul `setText("")`. Am folosit, de asemenea, butoane `CheckBox` pentru a permite utilizatorului actualizarea facilă a eventualei baze de date care stă în spatele formularului prin excluderea anumitor câmpuri determinată de butoane neselectate. Poziționarea componentelor în container s-a făcut folosind gestionarul `GridLayout`.

```
void faFormular(JFrame f)
{
    JPanel intrari = new JPanel();
    intrari.setBorder(
        new TitledBorder("Inregistrare utilizator"));
    // panou stang care tine checkbox-urile
    JPanel intraris = new JPanel();
    intraris.setLayout(new GridLayout(0, 1, 2, 1));
    JCheckBox numarCurent =
        new JCheckBox("Cod inregistrare");
    JCheckBox numeUtilizator =
        new JCheckBox("Nume utilizator");
```

```
JCheckBox numeCont = new JCheckBox("Nume cont");
JCheckBox parola = new JCheckBox("Parola identificare");
numeUtilizator.setSelected( true );
numeCont.setSelected( true );
intraris.add(numarCurent);
intraris.add(numeUtilizator);
intraris.add(numeCont);
intraris.add(parola);
// panou drept care tine zonele de text
JPanel intrarid = new JPanel();
intrarid.setLayout(new GridLayout(0, 1, 5, 3));
final CampNumar textNumarCurent=new CampNumar(4,5);
textNumarCurent.setFocusable(true);
final JTextField textNumeUtilizator=new JTextField(8);
final JTextFieldPersonalizat textNumeCont=
    new JTextFieldPersonalizat(8,6);
final JPasswordField textParola=new JPasswordField(8);
intrarid.add(textNumarCurent);
intrarid.add(textNumeUtilizator);
intrarid.add(textNumeCont);
intrarid.add(textParola);
// zona de text JTextArea pentru observatii
final Continut observatii=new Continut(2,5);
observatii.setWrapStyleWord(true);
observatii.setLineWrap(true);
observatii.setTabSize(222);
observatii.setMaximumSize(new Dimension(50,50));
// panou care contine eticheta si zona de text Continut
JPanel panouContinut=new JPanel();
panouContinut.setLayout(new BoxLayout(panouContinut,
    BoxLayout.Y_AXIS));
final JCheckBox Continut=
    new JCheckBox("Alte observatii");
Continut.setSelected( true );
panouContinut.add(Continut);
JScrollPane sp=new JScrollPane();
sp.setViewportView(observatii);
sp.setHorizontalScrollBarPolicy(JScrollPane.
    HORIZONTAL_SCROLLBAR_NEVER );
panouContinut.add(sp);
panouContinut.add(new JLabel(""));
panouContinut.setMaximumSize(new Dimension(20,20));

JPanel tot= new JPanel();
tot.setLayout(new BoxLayout(tot, BoxLayout.X_AXIS));
```

```

tot.add(intraris);
tot.add(intrarid);

intrari.setLayout(new BorderLayout());
intrari.add(tot, BorderLayout.WEST);
intrari.add(new JLabel(" "), BorderLayout.CENTER);
intrari.add(panouContinut, BorderLayout.EAST);
// constructie butoane
JPanel butoane = new JPanel();
butoane.setLayout (new FlowLayout(FlowLayout.LEFT));
JButton anuleaza = new JButton("Anuleaza");
JButton deacord = new JButton("De acord");
butoane.add(anuleaza);
butoane.add(deacord);
//ascultatori Butoane
anuleaza.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        textNumarCurent.setText("");
        textNumeUtilizator.setText("");
        textNumeCont.setText("");
        textParola.setText("");
        observatii.setText("");
    }
});
deacord.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        String NumarCurent=textNumarCurent.getText();
        String NumeUtilizator= textNume Utilizator.
            getText();
        String NumeCont=textNumeCont.getText();
        char[] parola = textParola.getPassword();
        String obs=observatii.getText();
        // eventuale actiuni ale utilizatorilor
    }
});
//panou mare
JPanel jsp=new JPanel();
jsp.add(intrari);
jsp.add(butoane);
f.getContentPane().add(jsp, BorderLayout.CENTER);
f.setVisible(true);
}

```

În procesul de personalizare a componentelor text, suprascriem pentru acestea, metoda având prototipul `protected Document createDefaultModel()` care

permite adăugarea drept model de date implicit, în procesul de creare a componentei, a propriului nostru model, ce reprezintă în fapt o implementare a clasei `Plain Document`. Singura metodă din această clasă care trebuie suprascrisă este public `void insertString(int deplasament, String sir, AttributeSet a)` throws `BadLocationException`, care este apelată de componentă de fiecare dată când un caracter sau un șir (spre exemplu, când lipim text componentei) este inserat în document. Prezentăm clasa `JTextFieldPersonalizat`, care este utilizată în exemplul precedent pentru obținerea de componente text având un număr fix de caractere care trebuie să fie litere sau caracterul spațiu. Aceste condiții sunt verificate folosind metodele `Character.isLetter(char c)` și `Character.isSpaceChar(char c)`, iar dacă acestea nu se îndeplinesc, se va emite un sunet de avertisment. Pentru construirea unei componente de acest tip se va folosi constructorul public `JTextFieldPersonalizat(int l, int init)`, unde `l` reprezintă lungimea zonei de text, iar `init` reprezintă numărul maxim de caractere.

```

// pentru a restrictiona numarul de caractere introduse
import javax.swing.*;
import javax.swing.text.*;
import java.awt.Toolkit;
import java.util.Locale;

public class JTextFieldPersonalizat extends JTextField {
    private Toolkit toolkit;
    int nrCaractere=0;
    public JTextFieldPersonalizat(int l,int init)
    {
        super(init);
        nrCaractere=l;
        toolkit = Toolkit.getDefaultToolkit();
    }

    public JTextFieldPersonalizat(int l)
    {
        super();
        nrCaractere=l;
        toolkit = Toolkit.getDefaultToolkit();
    }

    protected Document createDefaultModel() {
        return new DocumentPersonalizat();
    }

    protected class DocumentPersonalizat extends PlainDocument {
        public void insertString(int deplasament, String sir,
            AttributeSet a) throws BadLocationException

```

```

    {
        char[] sursa = sir.toCharArray();
        char[] rezultat = new char[sursa.length];
        int j = 0;
        if (JTextFieldPersonalizat.this.getText().
            length() < nrCaractere)
            for (int i = 0; i < rezultat.length; i++)
            {
                if (deplasament <= nrCaractere)
                    if (Character.isLetter(sursa[i]) ||
                        Character.isSpaceChar(sursa[i]))
                    { rezultat[j++] = sursa[i]; }
                    else { toolkit.beep(); }
                else { toolkit.beep(); }
                super.insertString(deplasament, new String(
                    rezultat, 0, j), a);
            }
    }
}

```

Pentru clasa `CampNumar` care permite introducerea unui număr fix de cifre, diferența față de `JTextFieldPersonalizat` constă în verificarea condiției `Character.isDigit(char c)` care returnează `true` în cazul în care `c` este o cifră.

În cazul `JTextArea` prezentăm în continuare implementarea unui model personalizat care va determina transferarea focusului în momentul apăsării tastei `Tab` (acest lucru nu se petrece, fiind implicit un inconvenient ce trebuie evitat), permițând și introducerea unui număr limitat de caractere:

```

protected class DocumentPersonalizat extends PlainDocument {
    public void insertString(int deplasament, String sir,
        AttributeSet a) throws BadLocationException
    {
        char[] sursa = sir.toCharArray();
        char[] rezultat = new char[sursa.length];
        int j = 0;
        if (Continut.this.getText().length() < numar
            Caractere)
            for (int i = 0; i < rezultat.length; i++)
            {
                if (deplasament <= numarCaractere)
                    if (Character.isLetterOrDigit(sursa[i]) || Character.
                        isSpaceChar(sursa[i]))
                    { rezultat[j++] = sursa[i]; }
                    else
                    { if (sursa[i] == '\t')
                        Continut.this.transferFocus(); }
            }
    }
}

```

```

        else { toolkit.beep(); }
    }
    super.insertString(deplasament, new String(rezultat,
        0, j), a);
}

```

Considerăm `Continut` numele componentei care extinde `JTextField` și numărul de caractere numărul de caractere maxim care poate fi introdus. Transferul focusului se face prin intermediul metodei `transferFocus()` pe care orice componentă grafică o moștenește din clasa `Component`.

În cazul componentei `JPasswordField`, citirea conținutului nu se face cu `getText()`, o astfel de încercare determinând un avertisment (eng. *warning*), deoarece această metodă a devenit învechită (eng. *deprecated*). Pentru a obține conținutul unei componente `JPasswordField` se folosește metoda `getPassword()` care returnează un tablou cu elemente `char`, care eventual poate fi criptat înainte de a fi stocat într-un fișier sau o bază de date, folosind API-urile pentru criptografie cu criptosistemele pe care Java le implementează (a se vedea pachetul `javax.crypto` și pachetele dependente de acesta).

Swing pune la dispoziția implementatorilor un sistem standard pentru a răspunde la dorințele utilizatorilor aplicației de a-și anula o anumită acțiune greșită (eng. *undo*), respectiv navigarea prin mulțimea stărilor mai vechi ale aplicației determinate de acțiunile utilizatorilor (eng. *undo/redo system*). Acest mecanism este implementat prin construcția componentelor text. Tot ceea ce trebuie să facem este să adăugăm componentei text, folosind metoda `addUndoableEditListener()`, un ascultător al evenimentelor `UndoableEditEvent` care sunt aruncate atunci când starea componentei se modifică (spre exemplu, s-a introdus sau s-a șters text). Ascultătorul folosește un gestionar care este o instanță a clasei `UndoManager` în care adaugă acțiunile înfășurate în evenimente. Gestionarul permite, mai apoi, prin intermediul metodelor `undo()`, `redo()` accesul la aceste acțiuni realizate de utilizatori și deci parcurgerea secvenței de stări ale aplicației.

Exemplul care urmează atașează un meniu pop-up componentei text observată din exemplul precedent, care se declanșează la apăsarea butonului drept al mouse-ului (pentru aceasta am adăugat un ascultător `MouseListener`). Meniul pop-up permite realizarea operațiilor de *undo/redo*, precum și *copy/cut/paste*. Pentru amănunte puteți urmări codul comentat, care este o completare la exemplul precedent:

```

import javax.swing.undo.*;
//...
final Continut observatii = new Continut(2, 5);
//...
final JPopupMenu pop = new JPopupMenu("Operatii");
final UndoManager gestionar = new UndoManager();
final JMenuItem si = new JMenuItem("Inapoi");

```

```

pop.add(si);
si.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_P,
    ActionEvent.CTRL_MASK));
si.setEnabled(false);
final JMenuItem su= new JMenuItem("Inainte");
pop.add(su);
su.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_I,
    ActionEvent.CTRL_MASK));
su.setEnabled(false);
si.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try { gestionar.undo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        si.setEnabled(gestionar.canUndo());
        su.setEnabled(gestionar.canRedo());
    }
});

su.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try { gestionar.redo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        si.setEnabled(gestionar.canUndo());
        su.setEnabled(gestionar.canRedo());
    }
});

observatii.getDocument().addUndoableEditListener(new
UndoableEditListener() {
    public void undoableEditHappened(UndoableEditEvent e) {
        gestionar.addEdit(e.getEdit());
        si.setEnabled(gestionar.canUndo());
        su.setEnabled(gestionar.canRedo());
    }
});

JMenuItem copie = new JMenuItem("Copie");
copie.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        observatii.copy();
    }
});

pop.add(copie);
copie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
    ActionEvent.CTRL_MASK));

```

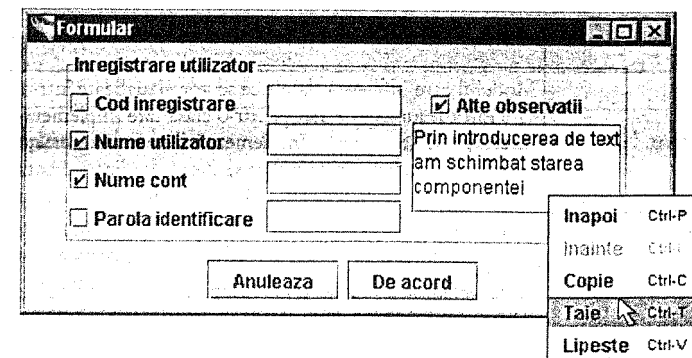
```

JMenuItem taie= new JMenuItem("Taie");
taie.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        observatii.cut();
    }
});
pop.add(taie);
taie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_T,
    ActionEvent.CTRL_MASK));

JMenuItem lipeste= new JMenuItem("Lipeste");
lipeste.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        observatii.paste();
    }
});
pop.add(lipeste);
lipeste.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V,
    ActionEvent.CTRL_MASK));

observatii.addMouseListener(new MouseAdapter()
{
    public void mousePressed(MouseEvent e)
    {
        int i=e.getClickCount();
        if (e.getButton()==3 && i==1)
        {
            pop.show(e.getComponent(), e.getX(), e.getY());
            pop.setVisible(true);
        }
    }
});
//...

```



Pe lângă aceste componente text relativ simple, programatorul mai are la dispoziție și altele mult mai complexe, și anume cele care au drept model `StyledDocument` prin care se oferă suport pentru formatele RTF sau HTML.

13.7.2. JTable

`JTable` este componenta grafică care permite operarea cu tabele. Clasa `JTable` din pachetul `javax.swing` extinde direct clasa `JComponent`, implementând interfețele `TableModelListener`, `TableColumnModelListener`, `ListSelectionListener`, `CellEditorListener` și `Scrollable`. `JTable` are asociate, de asemenea, trei modele: `TableModel`, `TableColumnModel` și `ListSelectionModel`, fiecare gestionând o anumită caracteristică a acestei componente. De asemenea, datorită complexității acestei componente, `JTable` are un întreg pachet dedicat, și anume `javax.swing.table`.

Componenta `JTable` este construită pe același principiu clasic în Swing, numit *model-delegat*, (eng. *model-delegate*), care reprezintă în fapt o despărțire a datelor de modul lor de vizualizare. Datele tabelului vor fi întotdeauna ținute într-o matrice sau într-o listă de vectori, sau o listă de liste, de fapt orice structură de date tabelară. Reprezentarea grafică a datelor (eng. *look*) este încapsulată împreună cu modul în care componenta reacționează la acțiunile utilizatorilor (eng. *feel*) într-un obiect, care poartă denumirea *delegate*, asociat fiecărei componente grafice. Componenta joacă astfel rolul unui mediator între vizualizare și datele care reprezintă conținutul ce se vrea vizualizat. Pentru `JTable` obiectul *delegate* implicit are numele `TableUI` și se găsește în pachetul `javax.swing.plaf`, și poate fi schimbat prin adăugarea unui plug-in nou pentru schimbarea vizualizării și comportării la acțiunile utilizatorilor (a se vedea capitolul Look-and-Feel). Această clasă dă, cu alte cuvinte, pentru componenta `JTable` vizualizarea și comportamentul (eng. *look-and-feel*). Datele ce se doresc vizualizate sunt la rândul lor înfășurate într-un model care prin sistemul de ascultători actualizează automat vizualizarea în urma modificărilor făcute în structura de date. Spre exemplu, atunci când vrem să ștergem o înregistrare dintr-o tabelă, o ștergem din structura care ține datele și atenționăm componenta `JTable` care o înfășoară să se reactualizeze.

Prezentăm în continuare cele trei modele ale unei instanțe `JTable`, împreună cu descrierile corespunzătoare:

Model	Funcționalitate
<code>TableModel</code>	Modelul care înfășoară datele ce se vor vizualiza într-o tabelă este întotdeauna gestionat printr-o clasă care implementează interfața <code>TableModel</code> . Implementările acestei interfețe specifică structura de date care se vrea vizualizată printr-o tabelă, precum și modalitățile de mănuire a acestei structuri.
<code>TableColumnModel</code>	Este folosit pentru a gestiona instanțe ale clasei <code>TableColumn</code> care reprezintă coloane ale tabelului <code>JTable</code> , oferind control asupra ordinii, selectării coloanelor și mărimii marginilor.
<code>ListSelectionModel</code>	Este folosit pentru a oferi mai multe moduri de selecție, și anume: un singur element, un singur interval și mai multe intervale.

Clasa `TableColumn` este responsabilă pentru vizualizarea unei coloane din tabela curentă. Fiecare instanță `TableColumn` are asociată câte o clasă care gestionează desenarea celulelor coloanei, editarea celulelor coloanei, desenarea capului de coloană, acestea permițând personalizarea fiecărei coloane în parte. Dacă aceste clase asociate coloanelor sunt setate `null`, se va folosi mecanismul implicit de desenare și editare atașat tabelului în întregime. Clasele care gestionează desenarea celulelor sunt instanțe ale interfeței `TableCellRenderer`, iar cele care gestionează editarea celulelor sunt instanțe ale clasei `CellEditor`. Pentru fiecare coloană putem specifica modalitatea în care ea reacționează la redimensionarea celorlalte coloane, când apar liniile despărțitoare între coloane și când nu, dimensiunea marginilor dintre linii și coloane, culoarea de fundal pentru celulele selectate, înălțimea celulelor, precum și lățimea coloanelor.

`JTable` oferă mai mulți constructori pentru realizarea tabelului, pe care îi vom sintetiza în tabelul următor datorită diversității lor:

Constructor	Explicație
<code>JTable()</code>	Construiește un tabel care este inițializat cu modelul de date implicit, modelul implicit pentru coloane și modelul implicit pentru selecție.
<code>JTable(int nrLinii, int nrColoane)</code>	Construiește un tabel cu dimensiunile <code>nrLinii</code> și <code>nrColoane</code> având celulele goale și drept model de date, <code>DefaultTableModel</code> .
<code>JTable(Object[][] liniiDate, Object[] numeColoane)</code>	Construiește un tablou pentru vizualizarea unei matrice <code>liniiDate</code> având numele coloanelor <code>numeColoane</code> .
<code>JTable(TableModel dm)</code>	Construiește un tabel având drept model de date <code>dm</code> , modelul implicit pentru coloane și modelul implicit pentru selecție.
<code>JTable(TableModel dm, TableColumnModel cm)</code>	Construiește un tabel care este inițializat cu modelul de date <code>dm</code> , modelul pentru coloane <code>cm</code> și modelul implicit pentru selecție.
<code>JTable(TableModel dm, TableColumnModel cm, ListSelectionModel sm)</code>	Construiește un tabel care este inițializat cu modelul de date <code>dm</code> , modelul pentru coloane <code>cm</code> și modelul <code>sm</code> pentru selecție.
<code>JTable(Vector liniiDate, Vector numeColoane)</code>	Construiește un tablou pentru vizualizarea unui Vector de elemente Vector <code>liniiDate</code> având numele coloanelor într-un Vector <code>numeColoane</code> .

Prezentăm în continuare un exemplu de tabelă simplă având toate modelele implicite, construită folosind unul dintre constructorii enumerați:

Nr. localitate	Localitate	Judet
1	alecsandru i. cuza	ias
2	andrieseni	ias
3	aroneanu	ias
4	baltati	ias

Am folosit metoda care urmează:

```
void faTabela(JFrame f)
{
    Object[][] liniiDate = {
        {"1", "alecsandru i. cuza", "iasi"},
        {"2", "andrieseni", "iasi"},
        {"3", "aroneanu", "iasi"},
        {"4", "baltati", "iasi"},
        {"5", "belcesti", "iasi"}
    };
    String[] numeColoane = {"Nr. localitate", "Localitate", "Judet"};
    JTable tabela = new JTable(liniiDate, numeColoane);
    JScrollPane jsp = new JScrollPane(tabela);
    f.getContentPane().add(jsp, BorderLayout.CENTER);
    f.setVisible(true);
}
```

Chiar dacă nu am creat explicit modele pentru date, coloane și selecție, acestea au fost create implicit, așa cum se poate vedea și din explicația constructorului folosit. Spre exemplu, drept model de date tabelă i s-a atașat implicit o instanță a clasei `DefaultTableModel` la care avem acces prin metoda `getModel()` din clasa `JTable`. Dacă dorim să atașăm un model unei componente `JTable`, putem folosi metoda `setModel()`.

Ne vom opri, în continuare, la modelul de date asociat unei componente `JTable`. După cum se observă din lista de constructori ai clasei `JTable`, în general, mai întâi se construiește modelul de date (modele, în general) și mai apoi acesta este folosit în construcția efectivă a tabelă. După cum am mai spus, interfața `TableModel` stă la baza modelelor de date pentru o tabelă și conține nouă metode care trebuie implementate de programatori, metode pe care le prezentăm în continuare:

Metodele interfeței	Explicația
<code>public void addTableModelListener (TableModelListener l)</code>	Adaugă un ascultător în lista de ascultători care va fi anunțat de fiecare dată când apare un eveniment model.
<code>public Class getColumnClass (int indiceColoana)</code>	Clasa cea mai generală a elementelor celulelor coloanei având indicele <code>indiceColoana</code> .
<code>public int getColumnCount()</code>	Returnează numărul de coloane.
<code>public String getColumnName (int indiceColoana)</code>	Returnează numele coloanei având indicele <code>indiceColoana</code> .
<code>public int getRowCount()</code>	Returnează numărul de linii.
<code>public Object getValueAt (int indiceLinie, int indiceColoana)</code>	Returnează elementul de la linia <code>indiceLinie</code> și coloana <code>indiceColoana</code> .

Metodele interfeței	Explicația
<code>public boolean isCellEditable (int indiceLinie, int indiceColoana)</code>	Returnează <code>true</code> dacă celula de la linia <code>indiceLinie</code> și coloana <code>indiceColoana</code> se vrea editabilă.
<code>public void removeTableModelListener (TableModelListener l)</code>	Șterge un ascultător din lista de ascultători care captează evenimentele modelului <code>TableModelEvent</code> .
<code>public void setValueAt (Object valoare, int indiceLinie, int indiceColoana)</code>	Setează valoarea <code>valoare</code> pentru linia <code>indiceLinie</code> și coloana <code>indiceColoana</code> .

Acestea specifică mănuierea structurii de date care va da conținutul tabelă și vor fi folosite în mod direct de componenta `JTable` în procesul de desenare a tabelă. Spre exemplu, se știe că în momentul desenării componenta `JTable` apelează foarte des metoda `getRowCount()` și deci programatorii trebuie să o implementeze eficient. Demonstrativ, vom rescrie metoda care desenează o tabelă într-o fereastră, din exemplul precedent, folosind de data aceasta un model personalizat. Drept rezultat se va obține aceeași vizualizare. Singura diferență este că tabelă nu mai este editabilă, datorită implementării pe care o vom da metodei `isCellEditable()`.

```
void faTabela(JFrame f)
{
    class ModelulMeu implements TableModel
    {
        Object[][] liniiDate = {
            {"1", "alecsandru i. cuza", "iasi"},
            {"2", "andrieseni", "iasi"},
            {"3", "aroneanu", "iasi"},
            {"4", "baltati", "iasi"},
            {"5", "belcesti", "iasi"}
        };
        String[] numeColoane = {"Nr. localitate", "Localitate", "Judet"};
        public void addTableModelListener(TableModelListener l)
        {
            /* nu dam implementare acestei metode */
        }
        public Class getColumnClass(int indiceColoana)
        {
            return liniiDate[1][indiceColoana].getClass();
        }
        /* in fapt String */
        public int getColumnCount()
        {
            return numeColoane.length;
        }
        public String getColumnName(int indiceColoana)
        {
            return numeColoane[indiceColoana];
        }
        public int getRowCount()
        {
            return liniiDate.length;
        }
        public Object getValueAt(int indiceLinie, int indiceColoana)
    }
}
```



```

    {
        return liniiDate[indiceLinie][indiceColoana];
    }
    public boolean isCellEditable(int indiceLinie, int indiceColoana)
    {
        return false;
    }
    public void removeTableModelListener(TableModelListener l)
    {
        //nu dam implementare acestei metode
    }
    public void setValueAt(Object Valoare, int indiceLinie, int
    indiceColoana)
    {
        /* nu dam implementare; folosind aceasta metoda se poate
        reactualiza continutul modelului (structurii de date) dupa
        ce introducem date in celulele vizualizarii */
    }
}
ModelulMeu m=new ModelulMeu();
JTable tabela = new JTable(m);
JScrollPane jsp=new JScrollPane(tabela);
f.getContentPane().add(jsp, BorderLayout.CENTER);
f.setVisible(true);
}

```

Swing pune la dispoziție două clase care reprezintă implementări implicite (eng. *default*) ale interfeței `TableModel` și care pot fi folosite, de asemenea, prin extindere pentru crearea modelelor de date proprii. Aceste clase sunt `AbstractTableModel` și `DefaultTableModel`. Prima este preferabil a fi extinsă, deoarece oferă libertate în alegerea structurii de date care stă la baza modelului. În cazul clasei `DefaultTableModel`, se știe că structura de date este un `Vector` (a se vedea clasa `Vector`) de elemente `Vector`.

În cazul în care extindem clasa `AbstractTableModel`, tot ceea ce trebuie să facem este să suprascriem trei metode care înfășoară structura aleasă pentru a ține datele tabelului într-un model. Cele trei metode care trebuie suprascrise sunt:

Prototipul metodei	Explicația metodei
<code>public int getRowCount()</code>	Folosită de componenta <code>JTable</code> corespunzătoare pentru a afla numărul de linii.
<code>public int getColumnCount()</code>	Folosită de componenta <code>JTable</code> corespunzătoare pentru a afla numărul de coloane.
<code>public Object getValueAt(int row, int column)</code>	Folosită de componenta <code>JTable</code> corespunzătoare pentru a afla conținutul celulelor pe care trebuie să-l vizualizeze.

Un model poate fi folosit ca un adaptor între `JTable` și structura de date pe care o vizualizează. Ca exemplu pentru clasa `AbstractTableModel`, prezentăm în continuare clasa `JDBCAdaptor` care reprezintă un adaptor pentru o tabelă stocată pe un server de baze de date, înfășurând rezultatul obținut în urma interogărilor efectuate.

Amănunte referitoare la baze de date și SQL puteți găsi în capitolul dedicat API-ului `JDBC`. Urmează câteva explicații asupra codului prezentat mai jos. Prin constructor se realizează conexiunea la tabela ce se dorește vizualizată. Folosind metoda `trimiteInterogare(String interogare)` vom stoca într-un vector de vectori datele tabelului, și anume rezultatul interogării păstrat într-un obiect `ResultSet`. Folosim clasa pe care Java o pune la dispoziție pentru a implementa vectori, și anume clasa `Vector` care reprezintă un vector având elemente `Object`, deci și obiecte `Vector`. Obiectul liniei va ține în final datele tabelului. De asemenea, în tabloul `numeColoane` obținem numele câmpurilor din tabelă, acest lucru ajutându-ne la atribuirea de denumiri sugestive coloanelor din tabela vizualizată. Pentru aceasta suprascriem, în continuare, metoda `getColumnName()` din interfața `Model`. Dacă nu am fi făcut această suprascriere, capul de tabel la afișare ar fi conținut numele câmpurilor din tabela `MySQL` (spre exemplu, `nr_loc` în loc de `Nr. localitate` așa cum va apărea după suprascriere). Urmează, mai apoi, cele trei metode ale clasei `AbstractTableModel`, despre care am vorbit mai sus, a căror implementare este extrem de simplă.

```

// aceasta clasa este o infasuratoare pentru tabela interogata
// (Model)
import java.util.Vector;
import java.sql.*;
import javax.swing.table.AbstractTableModel;
import javax.swing.event.TableModelEvent;
import javax.swing.*;
import java.awt.*;

public class JDBCAdaptor extends AbstractTableModel
{
    // variabile legate de SQL
    static Connection conexiune;
    static Statement stare;
    static ResultSet rezultat;
    static ResultSetMetaData meta;
    //variabile in care tinem datele tabelului
    String[] numeColoane = {};
    Vector linii = new Vector();
    //variabile legate de actualizare
    static boolean facemActualizare=false;
    static boolean editabila;
    //constructorul adaptorului
    public JDBCAdaptor(String bazaDate) { //stabilesc
    conexiunea cu baza de date
    try {
        Class.forName("org.gjt.mm.mysql.Driver").newInstance();
        String url="jdbc:mysql://localhost/"+bazaDate;

```

```

        conexiune = DriverManager.getConnection(url, "", "");
        stare = conexiune.createStatement();
    }
    catch(Exception ex) {
        System.err.println("Eroare la conectare");
        System.err.println(ex);
    }
}
//execut interogarea "intrebare" catre tabela curenta si
//culeg rezultatul in vectorul de vectorul linii
public void trimiteInterogare(String intrebare) {
    if (conexiune == null || stare == null) {
        System.err.println(
            "Nu avem asupra cui sa facem interogarea");
        return;
    }
    try {
        rezultat = stare.executeQuery(intrebare);
        meta = rezultat.getMetaData();
        int numarColoane = meta.getColumnCount();
        numeColoane = new String[numarColoane];
        // obtinem numele coloanelor
        for(int c = 0; c < numarColoane; c++) {
            numeColoane[c] = meta.getColumnLabel(c+1);
        }
        /* luam liniile tabelii de pe server si le
        introducem in tabloul de vectori */
        linii = new Vector();
        int incep=1;
        while (rezultat.next()) {
            incep++;
            Vector l = new Vector();
            for (int i = 1; i <= getColumnCount(); i++) {
                l.addElement(rezultat.getObject(i));
            }
            linii.addElement(l);
        }
    }
    catch (SQLException ex) {System.err.println(ex);
}

public String getColumnName(int column)
{
    String c="";
    if (numeColoane[column] != null)
        if (numeColoane[column].indexOf("nr_loc")!=-1)
            c="Nr. localitate";
        else if (numeColoane[column].indexOf

```

```

        ("localitate")!=-1)
            c="Localitate";
        else if (numeColoane[column].indexOf("plasa")!=-1)
            c="Plasa";
        else
            if (numeColoane[column].indexOf("judet")!=-1)
                c="Judet ";
            return c;
        }

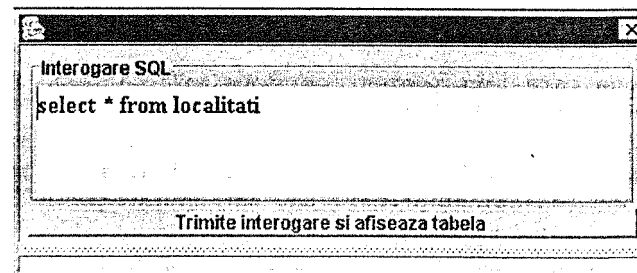
// Implementarea efectiva a adaptorului
public int getColumnCount() {
    return numeColoane.length; //numar coloane
}

public int getRowCount() {
    return linii.size(); //numar linii
}

public Object getValueAt(int linie, int coloana) {
    Vector row = (Vector)linii.elementAt(linie);
    return row.elementAt(coloana);
    //continutul celulelor
}
}

```

O dată modelul creat, printr-o metodă public void setModel (TableModel dataModel) acesta se poate asocia unui obiect JTable. O altă metodă este folosirea constructorului de forma public JTable (TableModel dm) prin care se creează o reprezentare vizuală a unei table având drept conținut datele din model, care se poate mai apoi introduce într-o componentă JScrollPane pentru a se putea derula.



Interfața grafică constă din două panouri încadrate într-un JSplitPane care le delimitează, unul în care se introduce interogarea într-o zonă de text și se poate apăsa un buton pentru a se obține vizualizarea rezultatului, iar celălalt este panoul în care se va afișa rezultatul, așa cum se poate vedea în figura următoare. Apăsarea butonului determină crearea modelului tablei, care este folosit mai apoi pentru a crea un obiect

JTable afișat într-un JPanel (comentariile care însoțesc codul sunt sugestive). Cea mai mare parte a codului este dedicată formătărilor interfeței grafice.

```
//interfata cu utilizatorul
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class Interogari extends JDialog {
    JPanel panouJos = new JPanel();
    JPanel panouSus = new JPanel();
    JSplitPane despartitor = new JSplitPane();
    JButton butont = new JButton();
    JTextArea text;
    Dimension marimeEcran;

    // constructor care initializeaza prezentarea
    public Interogari() {
        super();
        try {
            marimeEcran =
                Toolkit.getDefaultToolkit().getScreenSize();
            setBounds(marimeEcran.width/2-200,
                marimeEcran.height/2-100, 350,100);
            initDialog();
            pack();
        }
        catch(Exception ex) {      ex.printStackTrace();    }
    }

    //metoda care face prezentarea tabelii
    void initDialog() throws Exception {
        butont.setBorder(BorderFactory.createRaisedBevelBorder());
        butont.setText("Trimite interogare si afiseaza tabela");
        butont.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {try {
                JDBCAdaptor dt=
                    new JDBCAdaptor("starea_civila");
                //creare adaptor
                dt.trimiteInterogare(text.getText()); // initializarea
                // structurii modelului
```

```
despartitor.add(new JScrollPane(new
    JTable(dt)), JSplitPane.BOTTOM);
    //vizualizare
    }
    catch(Exception exe)
    {   System.out.println("Nu pot realiza operatiile!"
        +exe.toString());    }
    });

    JPanel panouSQL=new JPanel();
    JPanel derulare=new JPanel();
    final JLabel eticheta = new JLabel("SQL:");
    text = new JTextArea("select * from localitati");
    derulare.setBorder(BorderFactory.createTitledBorder
        (BorderFactory.createEtchedBorder(),"Interogare SQL"));
    derulare.setPreferredSize(new Dimension(400,100));
    text.setFont(new Font("Serif", Font.BOLD, 16));
    text.setLineWrap(true);
    derulare.setLayout(new BorderLayout());
    derulare.add(text);

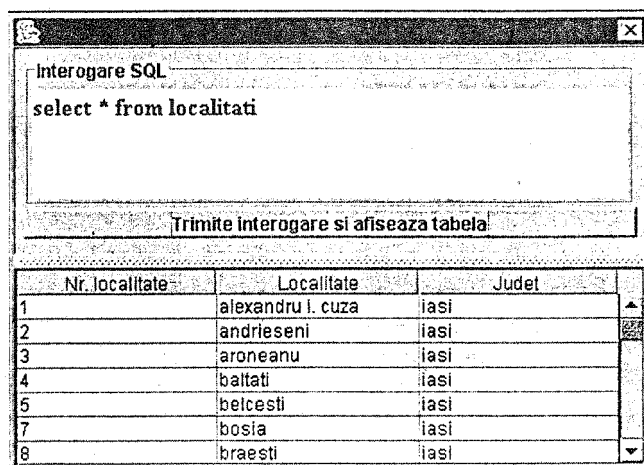
    panouSQL.setLayout(new BorderLayout());
    panouSQL.add(derulare,BorderLayout.NORTH);
    panouSQL.add(butont,BorderLayout.CENTER);

    panouSus.add(panouSQL, null);
    panouSus.setVisible(true);
    panouJos.setPreferredSize(new Dimension(410, 280));

    despartitor.setOrientation(JSplitPane.VERTICAL_SPLIT);

    despartitor.setTopComponent(panouSus);
    despartitor.add(panouJos, JSplitPane.BOTTOM);
    this.getContentPane().add(despartitor, BorderLayout.CENTER);
}

public static void main(String args[])
{
    Interogari j=new Interogari();
    j.setVisible(true);
}
```



În cazul clasei DefaultTableModel, accesul spre structura tabelară de date (după cum spuneam, implicit un vector de vectori) se realizează prin intermediul metodelor supraîncărcate de genul setDataVector() și getDataVector(). De asemenea, sunt puse la dispoziția utilizatorilor metode statice de genul convertToVector(), care realizează conversii ale tablourilor conținând instanțe ale clasei Object spre tablouri având elementele instanțe ale clasei Vector. Drept exemplu de model DefaultTableModel, vom adapta primul exemplu descris la începutul capitolului, folosind pentru aceasta unul dintre constructorii clasei DefaultTableModel, și anume cel care are prototipul public DefaultTableModel(Object[][] liniiDate, Object[] numeColoane), care folosește intern metoda setDataVector() pentru a inițializa structura tabelară:

```
void faTabela(JFrame f)
{
    Object[][] liniiDate = {
        {"1", "alecsandru i. cuza", "iasi"},
        //...
    };
    String[] numeColoane = {"Nr. localitate", "Localitate", "Judet"};
    DefaultTableModel m=new DefaultTableModel(liniiDate,
        numeColoane);
    JTable tabela = new JTable(m);
    JScrollPane jsp=new JScrollPane(tabela);
    f.getContentPane().add(jsp, BorderLayout.CENTER);
    f.setVisible(true);
}
```

JTable permite în parte personalizarea vizualizării tabelii fără a fi nevoie de schimbarea look-and-feel-ului. Spre exemplu, se pot adăuga iconuri ca fiind conținut

al celulelor, se poate modifica culoarea fundalului celulelor etc. Pentru aceasta trebuie suprascrisă clasa care realizează desenarea tabelii purtând numele TableCellRenderer. Prezentăm un exemplu de personalizare a desenării celulelor pentru una dintre tabelele realizate în acest capitol, prin care conținutul celulelor unei coloane este desenat folosind culori diferite și încadrat între două simboluri "*".

Nr. localitate	Localitate	Judet
9*	alecsandru i. cuza	iasi
4*	andrieseni	iasi
7*	aroneanu	iasi
4*	baltati	iasi

```
class DesenareCelule extends JLabel implements
    TableCellRenderer {
    public Component getTableCellRendererComponent(JTable table,
        // tabela
        Object value, // valoarea celulei
        boolean isSelected, // daca este selectată
        boolean hasFocus, // dacă are focus
        int row, // linia
        int column // coloana
    )
    {
        if (value!=null)
        {
            int nr;
            String valoare=value.toString();
            try
            { nr=Integer.parseInt(valoare); }
            catch (NumberFormatException e)
            { nr=1; }
            if (nr<12)
            setForeground(new Color(10*(12-nr),nr*7,13*nr));
            setText("*"+value.toString()+"*");
            setOpaque(isSelected ? true: false);
        }
        return this; // returneaza chiar aceasta eticheta
    }
}
```

Selectarea coloanei care se dorește personalizată se realizează prin secvența:

```
// Personalizare coloana
TableColumnModel tm=tabela.getColumnModel();
```

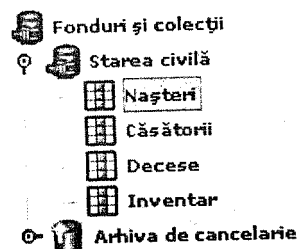
```

for(int i=0;i<tabela.getColumnCount();i++)
{
    TableColumn tc=tm.getColumn(i);
    String s=tabela.getColumnName(i);
    if(s.startsWith("Nr. "))
    {
        tc.setCellRenderer(new DesenareCelule());
    }
}

```

13.7.3. JTree

Componenta `JTree` permite vizualizarea structurilor arborescente. Clasa `javax.swing.JTree` este clasa de pornire pentru realizarea unui arbore. În afară de aceasta, un întreg pachet stă la baza implementării acestei componente în Swing, și anume `javax.swing.tree`. Cași în cazul `JTable`, o componentă `JTree` nu conține datele efective, ci doar oferă o vizualizare a datelor ținute de un model (o clasă care implementează `TreeModel`), eventual cu o formatare dată de un `UIDelegate`. Un exemplu simplu de arbore:



Metoda care determină această vizualizare este următoarea:

```

public JTree faArbore()
{
    DefaultMutableTreeNode StareaCivila;
    DefaultMutableTreeNode ArhivaDeCancelarie;
    DefaultMutableTreeNode radacina = new
        DefaultMutableTreeNode("Fonduri \u015Fi colec\u0163ii");
    radacina.add( StareaCivila = new
        DefaultMutableTreeNode("Starea civil\u0103") );
    radacina.add( ArhivaDeCancelarie = new
        DefaultMutableTreeNode("Arhiva de cancelarie") );
    StareaCivila.add( new
        DefaultMutableTreeNode("Na\u015Fteri") );
    StareaCivila.add( new
        DefaultMutableTreeNode("C\u0103s\u0103torii") );
}

```

```

StareaCivila.add( new
    DefaultMutableTreeNode("Decese") );
StareaCivila.add( new
    DefaultMutableTreeNode("Inventar") );
//...
final JTree arbore = new JTree(radacina);
DefaultTreeCellRenderer con =
    new DefaultTreeCellRenderer();
con.setLeafIcon(new ImageIcon("fl.gif"));
con.setOpenIcon(new ImageIcon("bl.gif"));
con.setClosedIcon(new ImageIcon("b2.gif"));
con.setFont(new Font("Verdana", Font.BOLD, 10));
arbore.setCellRenderer(con);
return arbore;
}

```

Un `JTree` prezintă nodurile arborelui unul sub altul pe verticală, câte un singur nod pe fiecare rând. Există un singur nod rădăcină, iar fiecare nod poate avea noduri copii (oricâți la număr) sau nu. Nodurile care nu au descendenți poartă numele de noduri frunză. Nodurile se pot extinde sau restrânge pentru a li se vedea sau nu descendenții. Operațiile de extindere/restrângere pot fi tratate prin atașarea de ascultători arborelui. Fiecare nod prezintă informația conținută într-o celulă a cărei înfățișare poate fi schimbată folosind o clasă care implementează `TreeCellRenderer`. De asemenea, modul în care utilizatorii vor edita această celulă poate fi controlat și personalizat prin intermediul unei clase care implementează `TreeCellEditor`.

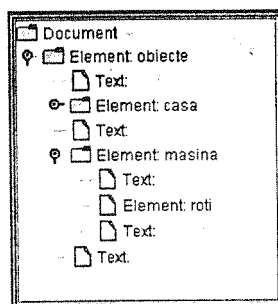
Mecanismul de selecție a unui nod al unui arbore `JTree` este asemănător cu mecanismul de selecție din `JList` și controlat prin intermediul unui model care este o implementare pentru `TreeSelectionModel`.

Interfața `TreeModel` stă la baza creării oricărui model de date pentru un `JTree`. Această interfață, prin clasele care o implementează, oferă posibilitatea mânăuirii structurii de date care dă conținutul unui `JTree`. Avem la dispoziție două modalități pentru a construi un model de date. Prima ar fi implementarea directă a interfeței `TreeModel`. Avantajul acestei modalități ar fi acela al unei mai mari lejerități în alegerea structurilor de date care vor fi alese drept noduri, implicit nodurile fiind instanțe ale clasei `Object`.

A doua ar fi extinderea clasei `DefaultTreeModel` care reprezintă implementarea implicită pe care Swing o dă interfeței `TreeModel`. În acest caz, nodurile vor fi instanțe ale unei clase care implementează interfața `TreeNode` sau ale unei clase care extinde această interfață, și anume `DefaultMutableTreeNode`. Alegerea uneia sau alteia dintre modalități rămâne la latitudinea programatorului.

Vom încerca în continuare să dăm o reprezentare vizuală unui document XML, căruia îi corespunde întotdeauna o structură arborescentă (numită arbore DOM) care-i descrie conținutul. Mai multe informații despre XML (inclusiv specificația) puteți găsi la adresa: www.w3c.org/xml. În urma parsării unui document XML

folosind DOM (eng. Document Object Model), structura arborescentă rezultată va fi o instanță a clasei `org.w3c.dom.Document`. Putem spune că clasa care implementează `TreeModel` înfășoară (eng. *wrap*) această structură arborescentă ce se dorește a fi vizualizată. Pentru aceasta trebuie să suprascriem toate metodele interfeței `JTreeModel` (7 la număr) într-o clasă care implementează această interfață. Aceste metode arată cum se poate obține nodul rădăcină al structurii de date, un anumit nod din structură, numărul de descendenți ai unui nod, când un nod este frunză sau nu, anunțarea modelului despre o schimbare în structura de date și adăugarea și ștergerea de ascultători ai modelului. Pentru amănunte puteți consulta API-ul DOM și capitolul dedicat procesării documentelor XML.



```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<obiecte>
```

```
<casa>
```

```
<masa>
```

```
<picioare numar="4" />
```

```
</masa>
```

```
<scaun>
```

```
</scaun>
```

```
</casa>
```

```
<masina>
```

```
<roti numar="4" />
```

```
</masina>
```

```
</obiecte>
```

Am prezentat documentul XML a cărui vizualizare am realizat-o. Nu am tratat tipurile de elemente care sunt găsite în timpul parsării, pentru a nu complica exemplul.

```
import java.io.*; //pentru intrari-iesiri
import org.w3c.dom.*; //DOM
import javax.xml.parsers.*; //pentru parserul DOM
import javax.swing.*; //Swing
import javax.swing.tree.*; //pentru JTree
import javax.swing.event.*; //pentru evenimente SWING
import javax.swing.border.*; //pentru bordere
```

```
import java.awt.*; //pentru AWT.
import java.awt.event.*; //pentru evenimente AWT
```

```
public class PanouArboreDom extends JPanel implements Constants
{
```

```
    DocumentBuilderFactory factory=DocumentBuilderFactory.
        newInstance();
    org.w3c.dom.Document document;//un document DOM
    JTree tree;
    JScrollPane treeView;
    InfasuratoareDOM model;
```

```
    public PanouArboreDom(String s)
```

```
    {
```

```
        // cream borderul
```

```
        EmptyBorder eb = new EmptyBorder(1,1,1,1);
```

```
        BevelBorder bb = new BevelBorder(BevelBorder.LOWERED);
```

```
        CompoundBorder cb = new CompoundBorder(eb,bb);
```

```
        this.setBorder(new CompoundBorder(cb,eb));
```

```
        //cream un arbore DOM din documentul
```

```
        try
```

```
        {
```

```
            DocumentBuilder builder=factory.newDocumentBuilder();
```

```
            document=builder.parse(new File(s));
```

```
            document.getDocumentElement().normalize();
```

```
            model = new InfasuratoareDOM();
```

```
            tree = new JTree(model);
```

```
            tree.getSelectionModel().setSelectionMode(
                TreeSelectionModel.SINGLE_TREE_SELECTION);
```

```
            //cream panoul de derulare
```

```
            treeView = new JScrollPane(tree);
```

```
            this.setLayout(new BorderLayout());
```

```
            this.add("Center",treeView );
```

```
        }
```

```
        catch (Exception pce)
```

```
        {
```

```
            System.out.println("Eroare"+pce);
```

```
        }
```

```
    } // constructor
```

```
//infasoara structura arborescenta rezultata in urma
//parsarii documentului XML
```

```

public class InfasuratoareDOM implements TreeModel
{
    //se da implementare metodelor din interfata TreeModel

    //returneaza radacina
    public Object getRoot() {
        return new InfasuratoareNOD(document);
    }

    //returneaza true daca nodul dat este frunza
    public boolean isLeaf(Object unNod) {
        InfasuratoareNOD nod = (InfasuratoareNOD) unNod;
        if (nod.numarCopii() > 0) return false;
        return true;
    }

    //returneaza numarul de copii pentru un nod dat
    public int getChildCount(Object parinte) {
        InfasuratoareNOD nod = (InfasuratoareNOD) parinte;
        return nod.numarCopii();
    }

    //returneaza copilul cu indicele nr
    public Object getChild(Object parinte, int nr) {
        InfasuratoareNOD nod = (InfasuratoareNOD) parinte;
        return nod.copilul(nr);
    }

    //returneaza indicele copilului copil pentru parintele
    //parinte
    public int getIndexOfChild(Object parinte, Object copil) {
        InfasuratoareNOD nod = (InfasuratoareNOD) parinte;
        return nod.index((InfasuratoareNOD) copil);
    }

    public void valueForPathChanged(TreePath path, Object
    newValue) {
        //nu scriem nimic.daca am face modificari in arbore pe
        //calea data de path
        System.out.println("modificat");
        InfasuratoareNOD aNode = (InfasuratoareNOD) path.
        getLastPathComponent();
        org.w3c.dom.Node sampleData = aNode.NodDOM;
        sampleData.setNodeValue((String)newValue);
    }
}

```

```

//pentru a adauga ascultatori personalizati
public void addTreeModelListener( TreeModelListener
listener ) { }

//pentru a sterge ascultatori personalizati
public void removeTreeModelListener( TreeModelListener
listener ) { }
}

//aceasta clasa infasura (wrap) un nod din org.w3c.dom.Node
//dand posibilitatea toString
class InfasuratoareNOD implements Constants
{
    boolean compress=true;
    int ind;
    //contine o variabila membru de tip Node si metode pentru
    //lucrul cu ea
    org.w3c.dom.Node NodDOM;

    //constructor care duce un nod DOM intr-un nod
    InfasuratoareNOD
    public InfasuratoareNOD(org.w3c.dom.Node nod) {
        NodDOM = nod;
    }

    //suprascriem toString din Object pentru a personaliza
    //ceea ce va aparea in celulele vizualizarii JTree
    public String toString() {
        String s = typeName[NodDOM.getNodeType()];
        String numeNOD = NodDOM.getNodeName();
        if (! numeNOD.startsWith("#")) {
            s += ": " + numeNOD;
        }

        if (NodDOM.getNodeValue() != null) {
            if (s.startsWith("ProcInstr"))
                s += ", ";
            else
                s += ": ";
        }

        String t = NodDOM.getNodeValue().trim();
        int x = t.indexOf("\n");
        if (x >= 0) t = t.substring(0, x);
        s += t;
    }
}

```

```

    }
    return s;
}

//in continuare se definesc o serie de metode
//care usureaza accesul la modurile arborelui

// returneaza indexul unui copil
public int index(InfasuratoareNOD copilul) {
    int numar = numarCopii();
    for (int i=0; i<numar; i++) {
        InfasuratoareNOD n = this.copilul(i);
        if (copilul == n) return i;
    }
    return -1; // in caz ca nu
}

//da numarul de copii pentru un nod DOM
public int numarCopii()
{
    return NodDOM.getChildNodes().getLength();
}

//returneaza copilul cu un anumit index pentru un nod DOM
public InfasuratoareNOD copilul(int indexDeCautare) {
    org.w3c.dom.Node nod =
        NodDOM.getChildNodes().item(indexDeCautare);
    return new InfasuratoareNOD(nod);
}
}

```

De observat că `TreeModel` ar putea fi folosită ca o metodă de a crea și ține structuri arborescente, fără a le da o reprezentare vizuală.

13.8. Meniuri și bare de unelte

13.8.1. Meniuri

Meniurile reprezintă modalitatea standard folosită în interfețele grafice pentru a permite utilizatorului aplicației alegerea unei opțiuni care generează o acțiune din mai multe posibile. Într-o aplicație, meniurile apar fie grupate într-o bară de meniu, fie ca meniuri pop-up care se vizualizează doar în momentul apăsării unui buton al mouse-ului.

Un meniu poate avea drept opțiuni alte submeniuri. Este indicat să nu se depășească trei niveluri în adâncime și, de asemenea, ca un meniu să nu conțină mai mult de șapte opțiuni pentru o bună funcționare a aplicației.

`JMenuBar` reprezintă un container pentru mai multe elemente `JMenu` așezate într-o bară orizontală în partea superioară a unei ferestre. Pentru a adăuga elemente `JMenu` unui meniu bară folosim metode cum ar fi `add(JMenu meniu)`. `JMenuBar` folosește `DefaultSingleSelectionModel` drept model pentru selecție, ceea ce determină ca un singur element `JMenu` să fie selectat la un moment dat. Deoarece `JMenuBar` moștenește `JComponent`, poate fi plasat oriunde în interiorul unui container. Pentru a adăuga meniu bară unei aplicații se folosește metoda `setJMenuBar()`, aplicabilă tuturor containerelor care implementează `RootPaneContainer` (a se vedea capitolul dedicat containerelor de bază).

Operații Editare Unelte Opțiuni Ajutor

`JPopupMenu` reprezintă o mică fereastră prezentând o coloană de elemente care sunt opțiuni din care se așteaptă ca utilizatorul să aleagă una. Adăugarea sau ștergerea de elemente se realizează folosind metoda `add()`, respectiv `remove()`. Folosind metoda `show()`, putem determina ca un meniu pop-up să devină vizibil la coordonatele specificate ale unei componente, după ce în prealabil a fost creat. Această componentă se numește apelant (eng. *invoker*) și se poate schimba folosind un apel `setInvoker(Component apelant)`. `JPopupMenu` detectează dacă componenta apelantă este un container de bază sau nu, comportându-se în consecință. Un exemplu complet de utilizare a meniurilor pop-up aveți în capitolul dedicat componentelor text.

Copie	Alt-C
Taie	Alt-T
Lipește	Alt-V
Sterge	Alt-S

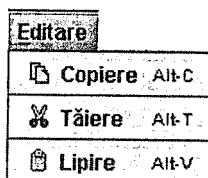
`JMenuItem` reprezintă o opțiune dintr-un meniu și este elementul de bază din compoziția meniurilor. Acestui element, pe lângă text, i se pot adăuga drept conținut iconuri. De asemenea, fiecărei opțiuni (un element `JMenuItem`) i se poate asocia o tastă (eng. *mnemonic*) folosind metoda `setMnemonic()` moștenită din clasa `AbstractButton`. Dacă în interiorul unei opțiuni litera care dă tasta asociată se regăsește, atunci va fi subliniată prima apariție a ei. Apăsarea tastei `Alt` (depinde de look-and-feel) împreună cu aceasta determină activarea opțiunii.

Operații	Editare	Unelte	Opțiuni	Ajutor
			Prezentare ▶	☐ Compiere
			Tiparire ▶	☒ Kunststoff
				☐ Metouia
				☐ ThreeD
				☐ Next

De asemenea, unei opțiuni meniu îi putem atașa un accelerator care va apărea ca text micșorat în partea dreaptă a elementului `JMenuItem`. Diferența dintre mnemonic și accelerator este aceea că un accelerator poate fi folosit chiar dacă meniul care conține opțiunea nu este vizibil la momentul curent, singura condiție necesară fiind aceea ca fereastra care conține meniul să fie activă. Atașarea acceleratorului se realizează folosind un apel de genul: `element.setAccelerator( KeyStroke.getKeyStroke( KeyEvent.VK_A, KeyEvent.CTRL_MASK, false),` ceea ce va determina ca opțiunea să se activeze prin apăsarea combinației de taste `Ctrl+A`.

Pentru a grupa opțiunile ce se doresc vizualizate într-o bară de meniu se folosește clasa `JMenu` care, surprinzător, extinde `JMenuItem`. Putem privi `JMenu` ca pe un `JMenuItem` care determină apariția unui meniu pop-up prin selectare. Instanțe ale clasei `JMenu` pot fi adăugate unui alt `JMenu` pentru a crea un submeniu al acestuia sau al meniului bară. Primul caz este ilustrat în figura precedentă, operația de adăugare determinând apariția unei săgeți care semnifică faptul că această opțiune are un submeniu. Al doilea caz este ilustrat în figura care urmează. Prin alegerea opțiunii Editare din bara de meniu se va desfășura un meniu pop-up conținând trei opțiuni. Acceleratorii pentru `JMenu` sunt dezactivați chiar dacă extinde clasa `JMenuItem`. Clasa `JMenu` conține un meniu pop-up, cel ce se va afișa în momentul selectării cu mouse-ul a numelui meniului. Acest meniu pop-up îl putem obține folosind metoda `getPopupMenu()`. Îi putem schimba locul afișării folosind metoda `setMenuLocation()`.

`JMenu` permite adăugarea de elemente `JMenuItem`, instanțe ale claselor implementând interfața `Action` (despre care vom discuta la finalul acestui subcapitol), alte componente sau șiruri de caractere folosind supraîncărcările metodei `add()`. De asemenea, între opțiunile unui meniu se pot adăuga linii despărțitoare folosind metoda `addSeparator()`, pentru a evidenția, spre exemplu, un grup de opțiuni. Apelul metodei `addSeparator()` determină crearea unei componente `JSeparator` reprezentând o zonă goală care este adăugată sub elementul curent al meniului. Posibilitatea de a adăuga alte componente drept opțiuni ale unui meniu (apelul de adăugare are forma `add(Component comp)`) poate fi folosită, spre exemplu, dacă dorim să adăugăm o componentă ceas sau oricare componentă personalizată.



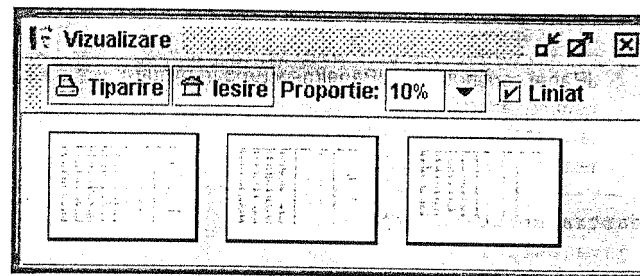
Clasa `JRadioButtonMenuItem` este folosită pentru adăugarea de butoane radio drept opțiuni în meniuri. Butoanele pot fi selectate folosind metoda `setSelected()` și li se pot atașa ascultători de tipul `ActionListener` sau `ChangeListener`. Pentru a permite setarea unui singur buton la un moment dat, trebuie să grupăm butoanele folosind clasa `ButtonGroup`. Un exemplu complet referitor la acest subiect găsiți în capitolul dedicat Look-and-feel-ului. De asemenea, pentru a putea adăuga căsuțe selectabile putem folosi clasa `JCheckBoxMenuItem`.

Pentru a intercepta setarea de către utilizator a unei opțiuni dintr-un meniu, de obicei atașăm acesteia un ascultător `ActionListener`. O altă alternativă ar fi folosirea instanțelor clasei `Action`, care permit crearea de elemente opțiuni înglobând și modul de tratare a selectării lor. Această modalitate este folosită, spre exemplu, în cazul în care o aceeași acțiune se regăsește în același timp și într-un meniu, și într-o bară de unelte, permițând implementarea de cod pentru tratarea acesteia o singură dată.

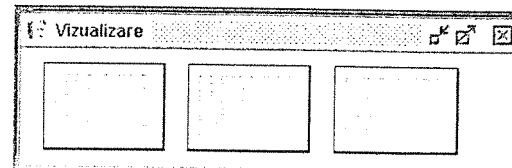
13.8.2. JToolBar

Bara de unelte (eng. *toolbar*) reprezintă o colecție de iconuri grupate într-o bară situată, în general, sub bara de meniu. Aceste iconuri sunt, de obicei, acceleratori pentru anumite opțiuni din meniul de bază al aplicației, aceasta deoarece este mai ușor să apăsăm un icon aflat deja pe suprafața de lucru decât să derulăm meniul. Bara de unelte poate fi folosită, așa cum îi spune și numele, și ca un mijloc de a oferi utilizatorilor diverse unelte cu care aceștia să lucreze pe parcursul rulării aplicației. Spre exemplu, în aplicațiile grafice, sunt puse la dispoziția utilizatorilor diverse unelte pentru desenare, alegerea iconurilor cu un design corespunzător determinând schimbarea cursorului mouse-ului și a funcționalității. O ultimă utilitate a barei de unelte ar fi pentru navigare. Un bun exemplu pentru acest caz ar putea fi un browser Web, trecerea de la o pagină la alta realizându-se prin iconurile din bara de navigare.

Pentru implementarea barei de unelte în Swing avem la dispoziție clasa `JToolBar`. Unei bare de unelte îi putem adăuga, folosind metoda `add()`, orice componentă, așa cum se poate vedea în figura care urmează, ilustrând o fereastră de vizualizare înaintea imprimării și în care avem 2 butoane `JButton`, un `JComboBox`, un buton `JCheckBox`.



De asemenea, folosind mouse-ul putem poziționa bara de unelte oriunde în interiorul ferestrei care o conține și chiar să o separăm de fereastră, unde o putem reintroduce folosind mouse-ul. Atunci când se află în exteriorul unei ferestre, bara de unelte este plasată într-o fereastră `JFrame`. Cele prezentate se pot observa în imaginile care urmează.

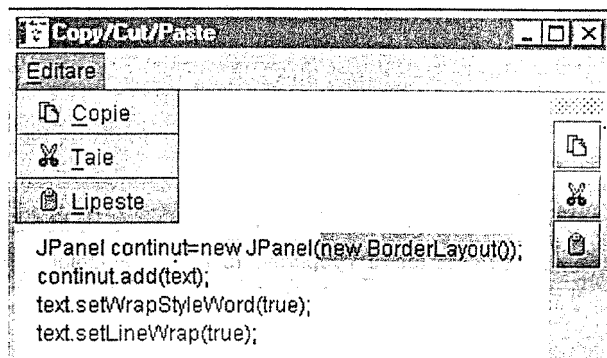




Între elementele unei bare de unelte se pot adăuga separatori folosind metoda `addSeparator()`. Drept exemplu de folosire a barelor de unelte aveți la dispoziție aplicația de la subcapitolul care descrie desenarea componentelor grafice.

Vom discuta, în continuare, despre interfața `Action`, care oferă o modalitate comună atât meniurilor, cât și barelor de unelte pentru a adăuga acțiuni reprezentate de butoane `JButton` în cazul `JToolBar` sau opțiuni `JMenuItem` în cazul `JMenu`, `JPopupMenu`, cu diverse funcționalități, folosind metoda `add(Action actiune)`. Interfața `Action` extinde interfața `ActionListener` și deci pune la dispoziția programatorilor metoda `actionPerformed()`, prin a cărei suprascriere se pot trata evenimentele `ActionEvent` generate de activarea acțiunilor. În practică este folosită clasa `AbstractAction` din `javax.swing`, care este o implementare abstractă a interfeței `Action`. Constructorii clasei `Action` permit vizualizarea de text sau iconuri ca reprezentare vizuală a acțiunii.

În aplicația care urmează vom folosi toate noțiunile acestui subcapitol:



```
/* Fereastra aplicatiei */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class FereastraAplicatie extends JFrame {
    //variabile de clasa
    static JMenuBar meniuBara;
    final JTextArea text=new JTextArea();
    Image img;
    int pozitie=230;
    //constructorul
```

```
public FereastraAplicatie() throws Exception {
    super("Copy/Cut/Paste");
    // pozitionam fereastra si-i setam dimensiunile
    Toolkit t=this.getToolkit();
    Dimension marimeEcran = Toolkit.getDefaultToolkit().
        getScreenSize();
    setBounds ( pozitie, pozitie, marimeEcran.width- 2*
        pozitie, marimeEcran.height/2-30);
    // setam iconul din coltul din stanga sus al ferestrei
    img=t.getImage("icon.jpg");
    this.setIconImage(img);
    this.getContentPane().setLayout(new BorderLayout());
    // construim partile componente
    faMeniu();
    faBaraUnelte();
    faContinut();
}
```

```
// construim continut
protected void faContinut() throws Exception
{
    JPanel continut=new JPanel(new BorderLayout());
    continut.add(text);
    text.setWrapStyleWord(true);
    text.setLineWrap(true);

    text.addMouseListener(new MouseAdapter()
    {
        // pentru a putea sa se deschida meniul
        public void mousePressed(MouseEvent e)
        {
            int i=e.getClickCount();
            if (e.getButton()==3 && i==1)
            {
                JPopupMenu pop=faMeniuPopup();
                pop.show(text, e.getX(), e.getY());
                pop.setVisible(true);
            }
        }
    });

    this.getContentPane().add(continut); // in acest panou vom
    //adauga continutul ferestrei
}

protected void faMeniu() {
    meniuBara = new JMenuBar(); // cream bara de meniu
    meniuBara.setOpaque(true);
```

```

JMenu editare= faMenuEditare(); //cream un meniu
                                //optiuni
editare.setMnemonic('E'); // definim shortcut (Alt+E)
menuBar.add(editare); // adaugam meniul optiuni la
                        // bara de meniu
setJMenuBar(menuBar);
}

// construim meniul editare
protected JMenu faMenuEditare() {
JMenu editeaza = new JMenu("Editare");
JMenuItem icopie=editeaza.add(new Copie("Copie",new
    ImageIcon("copie.gif")));
editeaza.addSeparator();
JMenuItem itaie=editeaza.add(new Taie("Taie",new
    ImageIcon("taie.gif")));
editeaza.addSeparator();
JMenuItem ilipeste=editeaza.add(new Lipeste("Lipeste",
    new ImageIcon("lipeste.gif")));
icopie.setMnemonic('C');
itaie.setMnemonic('T');
ilipeste.setMnemonic('L');
return editeaza;
}

JPopupMenu faMenuPopup()
{
    final JPopupMenu editeaza=new JPopupMenu("Editare");
    JMenuItem icopie=editeaza.add(new Copie("Copie",new
        ImageIcon("copie.gif")));
    JMenuItem itaie=editeaza.add(new Taie("Taie",new
        ImageIcon("taie.gif")));
    JMenuItem ilipeste=editeaza.add(new Lipeste("Lipeste",
        new ImageIcon("lipeste.gif")));
    icopie.setAccelerator(KeyStroke.getKeyStroke (KeyEvent.
        VK_C,ActionEvent.CTRL_MASK));
    itaie.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_T,
        ActionEvent.CTRL_MASK));
    ilipeste.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.
        VK_L, ActionEvent.CTRL_MASK));
    return editeaza;
}

void faBaraUnelte()
{

```

```

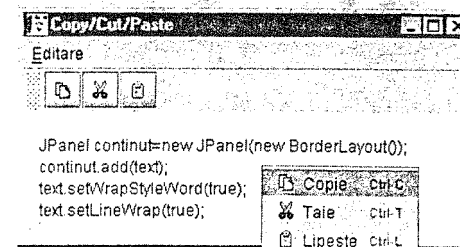
JToolBar tb = new JToolBar();
tb.add(new Copie("",new ImageIcon("copie.gif")));
tb.add(new Taie("",new ImageIcon("taie.gif")));
tb.add(new Lipeste("", new ImageIcon("lipeste.gif")));
this.getContentPane().add(tb,BorderLayout.NORTH);
}

// actiuni definite
class Copie extends AbstractAction
{
    Copie(String nume, ImageIcon icon)
    { super(nume, icon); }
    public void actionPerformed(ActionEvent e)
    { text.copy(); }
}

class Taie extends AbstractAction
{
    Taie(String nume, ImageIcon icon)
    { super(nume, icon); }
    public void actionPerformed(ActionEvent e)
    { text.cut(); }
}

class Lipeste extends AbstractAction
{
    Lipeste(String nume, ImageIcon icon)
    { super(nume, icon); }
    public void actionPerformed(ActionEvent e)
    { text.paste(); }
    public void iesire() {
        System.exit(0); // parasim aplicatia
    }
}

```



13.9. Dialoguri

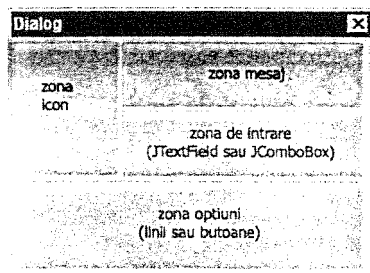
Dialogurile sunt necesare în cazul în care se dorește avertizarea utilizatorului asupra unui eveniment care s-a produs în aplicație sau în cazul în care se dorește validarea unor date introduse de utilizator. În acest ultim caz, fereastra de dialog va avea un buton de acceptare (eng. *accept*) care va putea fi apăsat doar dacă datele introduse de utilizator sunt valide. O altă utilizare a dialogurilor ar fi pentru a selecta dintr-o listă un set de elemente care vor fi adăugate în altă listă. Spre exemplu, prima listă poate reprezenta o mulțime de programatori dintr-o firmă, iar a doua listă poate fi formată doar din aceia care lucrează la un anumit proiect. Sunt și alte utilizări ale ferestrelor de dialog, dar este indicată folosirea lor cu grijă pentru a nu suprasolicita utilizatorul. Pentru a ocoli ferestrele de dialog care să prezinte utilizatorilor mesaje, acestea se pot afișa într-o bară de stare a ferestrei principale.

În unul din capitolele precedente am prezentat componenta `JDialog`, care reprezintă metoda standard de a crea ferestre de dialog și care permite realizarea de dialoguri complexe. Mai avem la dispoziție și alte componente, cum ar fi `JOptionPane`, pentru a realiza dialoguri simple și într-o formă standard. De asemenea, în Swing sunt implementate și câteva dialoguri standard personalizabile, cum ar fi cel pentru alegerea (eng. *choose*) unui fișier, și anume `JFileChooser`, respectiv pentru alegerea unei culori `ColorChooser`. Aceste componente le vom prezenta în continuare.

13.9.1. JOptionPane

Această clasă reprezintă o metodă facilă de a realiza dialoguri utile pentru a prezenta un mesaj, pentru a prezenta o întrebare utilizatorului și a aștepta introducerea răspunsului. Trebuie știut că fiecare dialog `JOptionPane` este modal (excepție fac dialogurile din ferestrele interne `JInternalFrame`), ceea ce înseamnă că firul de execuție al aplicației va fi blocat atât timp cât fereastra dialog este vizualizată. De asemenea, `JOptionPane` extinde direct clasa `JComponent` și nu `JDialog` așa cum ne-am aștepta, având rol de container intermediar, așa cum îi spune și numele, care poate fi așezat într-o fereastră `JFrame`, `JDialog` sau într-o fereastră interioară `JInternalFrame`.

Forma standard pe care o are un dialog `JOptionPane` este următoarea:



Se pot crea dialoguri folosind constructorii clasei `JOptionPane`, care sunt adăugate în `JDialog`-uri făcute mai apoi vizibile. De asemenea, se pot crea dialoguri folosind metode statice de forma `showXXXDialog()` ale clasei `JOptionPane`. Există patru tipuri de dialoguri, fiecare având metode specifice de afișare. Urmează prezentarea lor împreună cu metodele statice prin care se realizează activarea lor, respectiv explicațiile corespunzătoare.

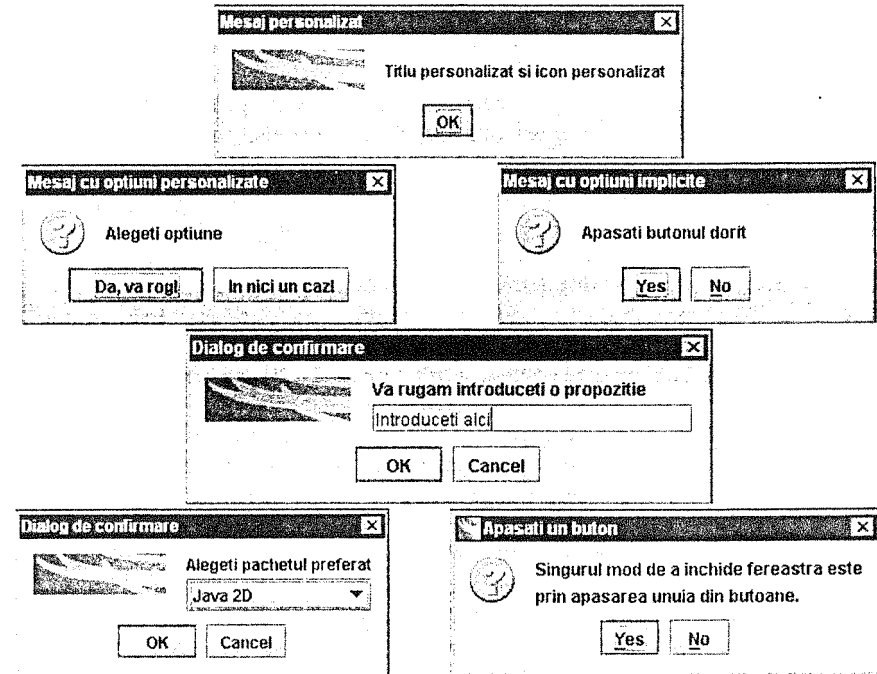
Tip mesaj	Metoda	Descriere
Mesaj	<code>showConfirmDialog()</code>	Această metodă afișează un dialog având câteva butoane și returnând un <code>int</code> reprezentând butonul apăsat. Metoda are patru supraîncărcări, permițând introducerea unei componente părinte, a unui mesaj, titlu, tip opțional, a tipului mesajului, a unui icon.
Confirmare	<code>showInputDialog()</code>	Această metodă afișează un dialog folosit pentru a obține date de la utilizator și returnează un <code>String</code> , dacă componenta de intrare este un câmp text sau un <code>Object</code> , dacă componenta de introducere este o listă sau un combobox. Sunt de asemenea șase supraîncărcări ale metodei cerând introducerea unei componente părinte, a unui mesaj, titlu, a tipului opțiunii, tipului mesajului, a unui icon, tablou de selecții posibile și a unui element selectat inițial. Mai sunt prezentate două butoane în zona opțională, și anume <code>OK</code> și <code>Cancel</code> .
Intrare	<code>showMessageDialog()</code>	Această metodă afișează un dialog cu un buton <code>OK</code> și nu returnează nimic. Sunt trei supraîncărcări ale metodei cerând introducerea unei componente părinte, a unui mesaj, titlu, tip de mesaj, icon.
Opțiune	<code>showOptionDialog()</code>	Această metodă afișează un dialog care poate fi personalizat mai mult decât celelalte dialoguri, returnând un index dintr-un tablou de opțiuni <code>Object</code> specificate. Există doar o metodă care primește o componentă părinte, un tip de opțiune, un tip de mesaj, un icon, un tablou de opțiuni <code>Object</code> și un obiect <code>Object</code> reprezentând focusul inițial.

Prezentăm în continuare explicațiile parametrilor care pot apărea în apelul metodelor de tipul `showXXXDialog()`. Pentru amănunte despre toate supraîncărcările acestor metode puteți să consultați `java-doc-ul`.

Parametri	Explicație
o componentă părinte	Reprezintă componenta care generează dialogul. Dacă părintele este <code>JFrame</code> , atunci dialogul va fi plasat într-un <code>JDialog</code> care va fi centrat relativ la fereastra părinte. Dacă părintele este <code>null</code> , atunci dialogul se va centra relativ la ecran.

Parametri	Explicație
un mesaj Object	Reprezintă un mesaj care va fi afișat în zona pentru mesaj. În general, este vorba de un String, așezat pe linii diferite folosind caracterul '\n'. Dacă mesajul nu este string, ci un alt obiect, acest lucru se tratează în felul următor : - Icon va fi vizualizat într-o etichetă JLabel. - Component va fi plasată în zona pentru mesaj. - Object [] vor fi plasate vertical într-o coloană verticală (rezultatul apelării toString()). - Object se va afișa textul rezultat ca la apelul toString().
un întreg care dă tipul mesajului	Poate avea una dintre valorile: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE. Această constantă spune look-and-feel-ului ce icon va fi afișat în zona destinată iconului. Iconurile implicite pentru tipurile de mesaje enumerate sunt:     întrebare, informații, avertisment, eroare
un întreg care dă tipul opțiunii	Poate avea una dintre valorile: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION. Această constantă spune ce butoane vor fi afișate în zona pentru opțiuni. Un alt set de opțiuni este reprezentat de ceea ce pot returna metodele showXXXDialog() prezentate în tabelul precedent. Acestea sunt CANCEL_OPTION, CLOSED_OPTION, NO_OPTION, OK_OPTION, YES_OPTION. De remarcat, CLOSED_OPTION este returnat când butonul de închidere de pe bara de text este apăsat.
un element Icon	Este iconul afișat în zona pentru icon. Dacă nu este specificat iconul ce se va afișa, va fi determinat de look-and-feel.
un tablou de obiecte Object	Putem specifica un tablou de obiecte care vor fi afișate în zona pentru opțiuni. Aceste obiecte pot fi șiruri String sau instanțe Icon care vor fi folosite pentru a fi decorate butoanele JButton, dar și instanțele Component care vor fi afișate toate pe un rând în zona pentru opțiuni. De asemenea, se va folosi toString() pentru a determina șirul care se va afișa.
o valoare inițială Object	Va specifica care componentă din zona pentru opțiuni va deține focusul.
un tablou de valori Object selectabile	Specifică un tablou de opțiuni din care se poate selecta una. Dacă tabloul are mai puțin de 12 opțiuni, se va folosi JList pentru prezentare, altfel se va folosi JComboBox.
un titlu String	Reprezintă titlul care se va afișa pe bara de titlu.

Prezentăm o aplicație demonstrativă care activează mai multe tipuri de dialoguri JOptionPane realizate prin metode de tipul showXXXDialog(). Ultimul dialog este realizat folosind constructorul pentru JOptionPane.



```

/*Aplicatia*/
import javax.swing.*;
import java.beans.*;
public class Aplicatia {
    public static void main( String[] args ) {
        try
        {
            UIManager.setLookAndFeel(
                "com.incors.plaf.kunststoff. KunststoffLookAndFeel");
            FereastraAplicatie fereastra =
                new FereastraAplicatie();
            fereastra.setVisible(true);
        }
        Icon icon=new ImageIcon("icon.jpg");

        // dialog mesaj
        JOptionPane.showMessageDialog(fereastra,
            "Titlu personalizat si icon personalizat", "Mesaj
            personalizat", JOptionPane.INFORMATION_MESSAGE, icon);
    }
}

```

```

        // dialog optiuni
Object[] optiuni = {"Da, va rog!", "In nici un caz!"};
int nr = JOptionPane.showOptionDialog(fereastra,
    "Alegeti optiune", "Mesaj cu optiuni personalizate",
    JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE,
    null, optiuni, optiuni[0]);
System.out.println("S-a ales butonul: "+optiuni[nr]);

// dialog confirmare
int numarul = JOptionPane.showConfirmDialog(fereastra,
    "Apasati butonul dorit", "Mesaj cu optiuni implicite",
    JOptionPane.YES_NO_OPTION);
System.out.println("S-a ales butonul: "+optiuni[numarul]);

//dialog de intrare
String iesire = (String)JOptionPane.showInputDialog(fereastra,
    "Va rugam introduceti o propozitie",
    "Dialog de confirmare",
    JOptionPane.INFORMATION_MESSAGE,
    new ImageIcon("icon.jpg"), null,
    "Introduceti aici");
System.out.println("S-a introdus: "+iesire);

// dialog de intrare folosind ComboBox
String[] pachete = new String[] {"AWT", "Swing",
    "Accessibility", "Java 2D", "Drag and Drop" };
String iesirea = (String)JOptionPane.showInputDialog(
    fereastra, "Alegeti pachetul preferat",
    "Dialog de confirmare",
    JOptionPane.INFORMATION_MESSAGE,
    new ImageIcon("icon.jpg"), pachete, "Java 2D");
System.out.println("S-a introdus: "+iesirea);

// dialog realizat prin constructor
final JOptionPane panouOptiuni = new JOptionPane(
    "Singurul mod de a inchide fereastra este\n"+
    "prin apasarea unuia din butoanele.\n" +
    "Mesaj cu obtiuni", JOptionPane.QUESTION_MESSAGE,
    JOptionPane.YES_NO_OPTION);
final JDialog dialog = new JDialog(fereastra,
    "Apasati un buton", true);
dialog.setContentPane(panouOptiuni);
panouOptiuni.addPropertyChangeListener(

```

```

new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        String prop = e.getPropertyName();
        if (dialog.isVisible() && (e.getSource() ==
            panouOptiuni) &&
            (prop.equals(JOptionPane.VALUE_PROPERTY) ||
            prop.equals(JOptionPane.INPUT_VALUE_
            PROPERTY)))
        {
            dialog.setVisible(false);
        } } };
dialog.pack();
dialog.setVisible(true);

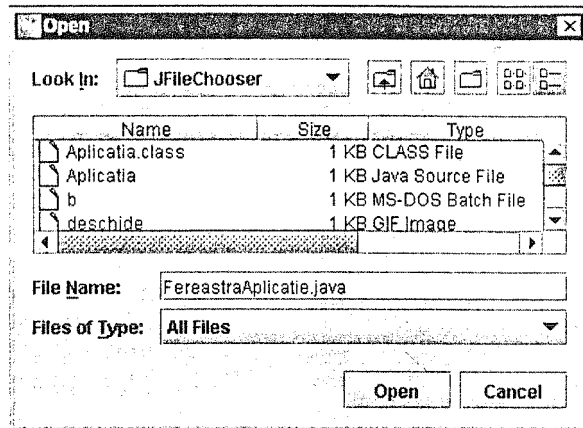
int valoare = ((Integer)panouOptiuni.getValue()).intValue();
if (valoare == JOptionPane.YES_OPTION) {
    System.out.println("S-a ales butonul <Da> ");
} else
    System.out.println("S-a ales butonul <Nu> ");
}
catch(Exception ex)
{
    System.out.println("Eroare"+ex.toString());
}
}

```

13.9.2. JFileChooser

Swing introduce un mod standard pentru navigarea în structura de directoare și selecția de fișiere, nemaifiind nevoiți să realizăm propriile noastre ferestre de dialog pentru aceste operațiuni des întâlnite. Este vorba de un `JList` și câteva butoane plus alte componente care permit introducerea de date, toate legate împreună pentru a forma o singură componentă atomică numită `FileChoose` oferind aceeași funcționalitate ca dialogurile fișier de pe platformele native. Pentru a seta directorul curent se poate folosi metoda `setCurrentDirectory(String sir)`. Pentru a vizualiza un dialog se pot folosi apeluri de genul `showOpenDialog()` pentru deschiderea unui fișier sau `showSaveDialog()` pentru salvarea unui fișier.

Deoarece inițializarea unei ferestre `FileChooser` necesită mult timp, este indicat ca operația de inițializare (crearea unei instanțe a variabilei `dialog JFileChooser` `dialog = new JFileChooser()`) să se realizeze într-un fir de execuție separat la



startarea aplicației. JFileChooser permite folosirea de filtre pentru a vizualiza într-un FileChooser numai anumite tipuri de fișiere. Pentru aceasta se extinde clasa `FileFilter` din pachetul `javax.swing.filechooser`, din care se suprascriu două metode:

Metoda	Explicație
<code>boolean accept(File f)</code>	Returnează <code>true</code> dacă se dorește vizualizarea fișierului <code>f</code> , altfel se va returna <code>false</code> .
<code>String getDescription()</code>	Returnează o descriere a filtrului folosit în JComboBox-ul din partea de jos a dialogului JFileChooser.

Pentru a gestiona filtrele JFileChooser pune la dispoziție câteva metode:

Metoda	Explicație
<code>addChoosableFileFilter(FileFilter f)</code>	Pentru a adăuga un alt filtru.
<code>removeChoosableFileFilter(FileFilter f)</code>	Pentru a șterge un filtru deja existent.
<code>setFileFilter(FileFilter f)</code>	Pentru a seta un filtru să fie activ (și îl adaugă dacă este necesar).

Implicit, JFileChooser folosește un filtru care permite vizualizarea tuturor fișierelor (eng. *all files*). Putem să ștergem acest filtru folosind secvența de comenzi:

```
FileFilter ft = dialog.getAcceptAllFileFilter();
dialog.removeChoosableFileFilter(ft);
```

Prezentăm o clasă șablon care poate fi simplu folosită pentru a implementa propriile noastre filtre:

```
class Filtru extends javax.swing.filechooser.FileFilter
{
```

```
private String descriere = null;
private String extensie = null;

public Filtru(String extensie, String descriere) {
    this.descriere = descriere;
    this.extensie = "." + extensie.toLowerCase();
}

public String getDescription() {
    return descriere;
}

public boolean accept(File f) {
    if (f == null)
        return false;
    if (f.isDirectory())
        return true;
    return f.getName().toLowerCase().endsWith(extensie);
}
}
```

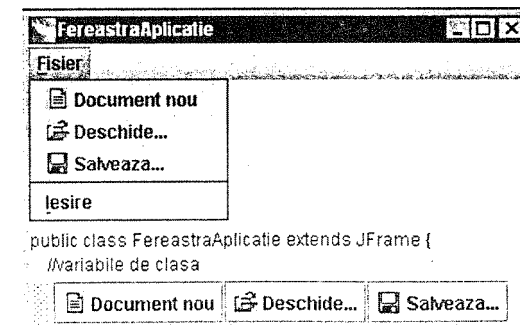
All Files

*.java
*.class
*.xml

```
Dialogul.setFileFilter(new Filtru("java", "*.java"));
Dialogul.setFileFilter(new Filtru("class", "*.class"));
Dialogul.setFileFilter(new Filtru("xml", "*.xml"));
```

A se observa că filtrul implicit nu a fost șters.

Prezentăm în continuare o implementare a meniului Fisier (eng. *file*), întâlnit în toate aplicațiile. În același timp se adaugă și o bară JToolBar ferestrei de bază.



```

protected void faMeniu() {
    meniuBara = new JMenuBar(); // cream bara de meniu
    meniuBara.setOpaque(true);
    JMenu Fisier= faMeniuFisier(); //cream un meniu
    // optiuni
    Fisier.setMnemonic('F'); // definim shortcut (Alt+F)
    meniuBara.add(Fisier); // adaugam meniul optiuni la
    // bara de meniu
    setJMenuBar(meniuBara);
}

// construim meniul optiuni
protected JMenu faMeniuFisier() {

    final JFileChooser Dialogul = new JFileChooser();
    Dialogul.setCurrentDirectory(new File("."));

    JMenu meniuFisier= new JMenu("Fisier");

    ImageIcon iconNou = new ImageIcon("nou.gif");
    Action actiuneNou =
        new AbstractAction("Document nou", iconNou)
    {
        public void actionPerformed(ActionEvent e)
        {
            zonaText.setText("");
        }
    };

    JMenuItem element = meniuFisier.add(actiuneNou);
    meniuFisier.add(element);

    ImageIcon iconDeschide = new ImageIcon("deschide.gif");
    Action actiuneDeschide= new AbstractAction
        ("Deschide...", iconDeschide)
    {
        public void actionPerformed(ActionEvent e)
        {
            if (Dialogul.showOpenDialog(FereastraAplicatie.this)
                != JFileChooser.APPROVE_OPTION)
                return;
            Thread fir = new Thread() {
                public void run() {
                    File fisier = Dialogul.getSelectedFile();
                    try

```

```

        {
            FileReader intrare = new FileReader(fisier);
            zonaText.read(intrare, null);
            intrare.close();
        }
        catch (IOException ex)
        {
            ex.printStackTrace();
        }
    });
    fir.start();
}

};

element = meniuFisier.add(actiuneDeschide);
meniuFisier.add(element);

ImageIcon iconSalveaza = new ImageIcon("salveaza.gif");
Action actiuneSalveaza = new AbstractAction(
    "Salveaza...", iconSalveaza)
{
    public void actionPerformed(ActionEvent e)
    {
        if (Dialogul.showSaveDialog(FereastraAplicatie.this)
            != JFileChooser.APPROVE_OPTION)
            return;
        Thread fir = new Thread() {
            public void run() {
                File fisier = Dialogul.getSelectedFile();
                try
                {
                    FileWriter iesire = new FileWriter(fisier);
                    zonaText.write(iesire);
                    iesire.close();
                }
                catch (IOException ex)
                {
                    ex.printStackTrace();
                }
            }
        };
        fir.start();
    }
};

element = meniuFisier.add(actiuneSalveaza);
meniuFisier.add(element);

```



```

menuFisier.addSeparator();

Action actiuneIesire = new AbstractAction("Iesire")
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
};

element = menuFisier.add(actiuneIesire);
element.setMnemonic('I');

JToolBar baraUnelte = new JToolBar();
JButton bNew = new JButton(actiuneNou);
baraUnelte.add(bNew);

JButton bOpen = new JButton(actiuneDeschide);
baraUnelte.add(bOpen);

JButton bSave = new JButton(actiuneSalveaza);
baraUnelte.add(bSave);

FereastraAplicatie.this.getContentPane().add(baraUnelte,
BorderLayout.NORTH);

return menuFisier;
}

```

13.9.3. JColorChooser

JColorChooser reprezintă o componentă standard pusă la dispoziția dezvoltatorilor și folosită pentru selecția unei culori. Fereastra JColorChooser este folosită în mod modal (blochează execuția aplicației). Permite trei modalități de alegere a culorilor Swatches, HSB și RGB. Putem vedea și alege din culorile deja selectate. O fereastră de dialog se activează folosind constructorul clasei JColorChooser, având drept parametri fereastra părinte, numele ferestrei, respectiv culoarea implicită prezentată la deschiderea dialogului.

```

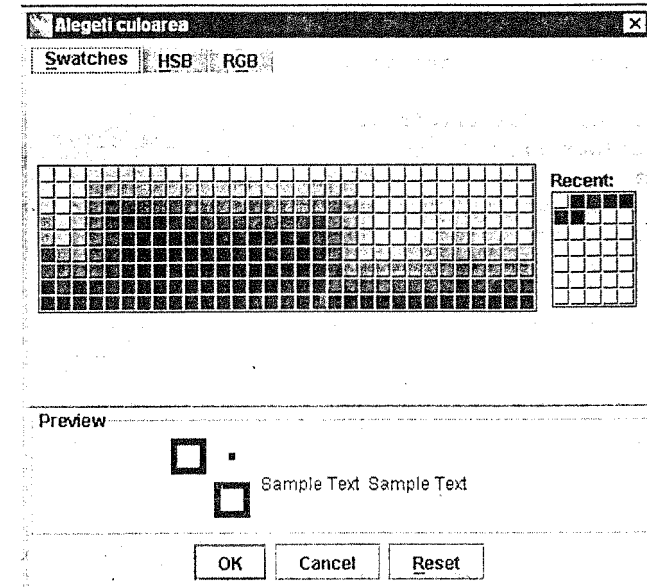
Color culoare = JColorChooser.showDialog(this, "Alegeti culoarea",
Color.red);
if (culoare != null)
{

```

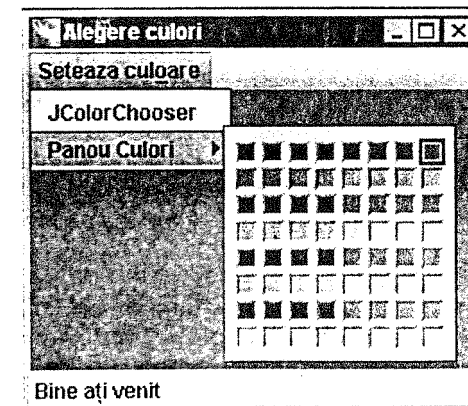
```

// actiune dupa alegerea culorii
continut.setBackground(culoare);
// in acest caz coloreaza background-ul folosind
// culoarea aleasa
}

```



Modul în care arată fereastra de dialog JColorChooser poate fi personalizat, dar nu vom insista asupra acestui lucru, vizualizarea acestui dialog părând a fi suficientă. Vom prezenta, în schimb, o metodă de a selecta culorile folosind un panou deschis la alegerea unei opțiuni dintr-un meniu.



```

class PaletaCulori extends JMenu
{
    protected Border neselectat;
    protected Border selectat;
    protected Border activ;

    protected Hashtable perechi;
    protected PanouCuloare casuta;

    public PaletaCulori(String nume) {
        super(nume);
        neselectat = new CompoundBorder( new MatteBorder(1, 1, 1, 1,
            getBackground()), new BevelBorder(BevelBorder.LOWERED,
            Color.white, Color.gray));
        selectat = new CompoundBorder( new MatteBorder(2, 2, 2, 2,
            Color.red), new MatteBorder(1, 1, 1, 1, getBackground()));
        activ = new CompoundBorder( new MatteBorder(2, 2, 2, 2,
            Color.blue), new MatteBorder(1, 1, 1, 1, getBackground()));

        JPanel panou = new JPanel();
        panou.setBorder(new EmptyBorder(5, 5, 5, 5));
        panou.setLayout(new GridLayout(8, 8));
        perechi = new Hashtable();

        int[] values = new int[] { 0, 128, 192, 255 };
        for (int r=0; r<values.length; r++) {
            for (int g=0; g<values.length; g++) {
                for (int b=0; b<values.length; b++) {
                    Color c = new Color(values[r], values[g], values[b]);
                    PanouCuloare pn = new PanouCuloare(c);
                    panou.add(pn);
                    perechi.put(c, pn);
                }
            }
        }
        add(panou);
    }

    public void seteazaCuloare(Color c) {
        Object obj = perechi.get(c);
        if (obj == null)
            return;
        if (casuta != null)
            casuta.setSelected(false);
        casuta = (PanouCuloare)obj;
    }
}

```

```

        casuta.setSelected(true);
    }

    public Color daCuloare() {
        if (casuta == null)
            return null;
        return casuta.daCuloare();
    }

    public void selecteaza() {
        // actiune dorita
        FereastraAplicatie.continut.setBackground(daCuloare());
    }

    class PanouCuloare extends JPanel implements MouseListener
    {
        protected Color m_c;
        protected boolean casuta;

        public PanouCuloare(Color c) {
            m_c = c;
            setBackground(c);
            setBorder(neselectat);
            String msg = "R "+c.getRed()+"", G "+c.getGreen()+
                ", B "+c.getBlue();
            setToolTipText(msg);
            addMouseListener(this);
        }

        public Color daCuloare() { return m_c; }

        public Dimension getPreferredSize() {
            return new Dimension(15, 15);
        }

        public Dimension getMaximumSize() { return
            getPreferredSize(); }
        public Dimension getMinimumSize() { return
            getPreferredSize(); }

        public void setSelected(boolean selected) {
            casuta = selected;
            if (casuta)
                setBorder(selectat);
            else
                setBorder(neselectat);
        }
    }
}

```

```

    }

    public boolean isSelected() { return casuta; }

    public void mousePressed(MouseEvent e) {}

    public void mouseClicked(MouseEvent e) {}

    public void mouseReleased(MouseEvent e) {
        seteazaCuloare(m_c);
        MenuSelectionManager defaultManager().
            clearSelectedPath();
        selecteaza();
    }

    public void mouseEntered(MouseEvent e) {
        setBorder(activ);
    }

    public void mouseExited(MouseEvent e) {
        setBorder(casuta ? selectat : neselectat);
    }
}

```

13.10. Componente pentru progres și derulare

JSlider, JScrollBar și JProgressBar sunt componente care ilustrează un domeniu interval și o valoare care evoluează între o valoare minimă și o valoare maximă reprezentând cele două capete ale intervalului. Interfața BoundedRangeModel reprezintă un model de date care definește o valoare întreagă ce reprezintă un indicator evoluând între o valoare minimă și una maximă. Această valoare are asociată o altă care reprezintă mărimea indicatorului. Valoarea nu poate fi niciodată mai mică decât valoarea minimă și mai mare decât valoarea maximă.

DefaultBoundedRangeModel reprezintă implementarea implicită a interfeței BoundedRangeModel. Acest model aruncă evenimente ChangeEvents conform specificației *JavaBeans*, atunci când o proprietate își schimbă valoarea. Toate cele trei componente descrise în continuare au implicit asociat drept model de date o instanță a clasei DefaultBoundedRangeModel.

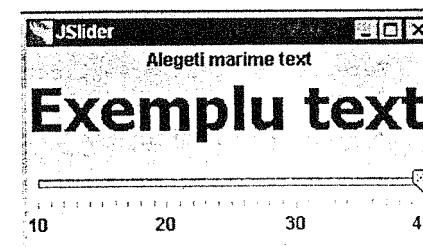
13.10.1. JSlider

JSlider reprezintă o metodă convenabilă pentru a alege o valoare numerică prin mutarea unui indicator pe suprafața unei rigle marcate. Componenta JSlider este foarte folositoare în cazul în care se cunoaște domeniul de valori pe care trebuie să le introducă utilizatorul, permițând acestuia alegerea facilă a uneia din valorile posibile. Valorile sunt evidențiate pe riglă folosind marcaje mici sau mari și cresc de la stânga spre dreapta și de jos în sus. Dacă poziția marcajelor mari corespunde cu a celor mici, acestea din urmă nu vor fi afișate. Marcajele mari pot fi întotdeauna etichetate folosind componente, iar implicit acestea sunt etichete JLabel care le prezintă valoarea.

Componenta JSlider are următoarele proprietăți:

Proprietăți	Explicații
orientation	Oferă posibilitatea de a orienta vertical sau orizontal componenta prin setarea acestei proprietăți cu valorile JSlider.HORIZONTAL sau JSlider.VERTICAL.
extent	Specifică numărul de valori sărite înainte și înapoi prin folosirea tastelor PageUp și PageDown.
minorTickSpacing	Spațiul dintre marcajele mari.
majorTickSpacing	Spațiul dintre marcajele mici.
paintTicks	Spune dacă se vor desena marcaje.
paintLabels	Setează/desetează afișarea etichetelor cu valori pentru marcajele mari.
paintTrack	Specifică dacă componenta este colorată în interior sau nu.
inverted	Pentru a schimba direcția incrementării.

Pentru fiecare proprietate avem metode de acces corespunzătoare. De fiecare dată când una din proprietăți își schimbă starea, conform specificației *JavaBeans* se va arunca un eveniment PropertyChangeEvent care încorporează proprietatea a cărei valoare s-a schimbat, putând astfel trata evenimentul produs. A se vedea secțiunea dedicată descrierii *JavaBeans*.



```

static JTextArea text;
//...
protected void faContinut() throws Exception
{

```

```

JPanel continut= new JPanel(); // in acest panou vom
adauga continutul ferestrei
this.getContentPane().add(continut, BorderLayout.CENTER);
int dis=10;
JLabel eticheta = new JLabel("Alegeti marime text",
JLabel.CENTER);
eticheta.setAlignmentX(Component.CENTER_ALIGNMENT);
JSlider slider = new JSlider(JSlider.HORIZONTAL, 10, 40, dis);
slider.addChangeListener(new AscultatorSlider());
slider.setMajorTickSpacing(10);
slider.setMinorTickSpacing(1);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
slider.setBorder(BorderFactory.createEmptyBorder
(0, 0, 10, 0));
Font f=new Font("Tahoma", Font.BOLD, 10);
text=new JTextArea();
text.setFont(f);
text.setForeground(new Color(128, 32, 128));
text.setBackground(eticheta.getBackground());
text.setText("Exemplu text");
text.setEditable(false);
continut.setLayout(new BorderLayout());
continut.add(eticheta, BorderLayout.NORTH);
continut.add(text, BorderLayout.CENTER);
continut.add(slider, BorderLayout.SOUTH);

```

```

class AscultatorSlider implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
        JSlider sursa = (JSlider)e.getSource();
        if (!sursa.getValueIsAdjusting()) {
            int fps = (int)sursa.getValue();
            Font f=new Font("Tahoma", Font.BOLD, fps);
            FereastraAplicatie.text.setFont(f);
            FereastraAplicatie.text.repaint();
        }
    }
}

```

13.10.2. JScrollBar

Reprezintă implementarea unei bare de derulare. Această componentă este des folosită ca parte integrantă a componentei JScrollPane în procesul de derulare a conținutului, dar poate fi folosită și separat, așa cum vom vedea în această secțiune. Utilizatorul

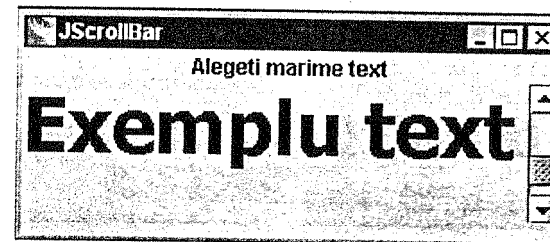
plimbă un indicator între două limite, una inferioară și una superioară (de fapt, alege o valoare dintr-un interval domeniu). Pentru acest lucru pot folosi și butoanele care se află la cele două extreme ale componentei. Zona componentei JScrollBar, care se află în exteriorul indicatorului și al butoanelor, poartă numele de zona de paginare și poate fi folosită la rândul ei pentru a muta indicatorul prin poziționarea cursorului mouse-ului. Deosebirea ei față de JSlider (aparent par a avea aceeași funcționalitate) constă în faptul că JSlider este folosită pentru a permite utilizatorului să aleagă o valoare.

Indicatorul are o anumită mărime, care este gestionată prin intermediul proprietății `visibleAmount`. Folosind metoda `setOrientation()` putem seta orientarea componentei folosind cele două constante pe care le avem la dispoziție: `JScrollBar.HORIZONTAL` și `JScrollBar.VERTICAL`. Proprietatea `unitIncrement` specifică pasul cu care se va face deplasarea indicatorului prin apăsarea butoanelor extreme. De asemenea, pasul cu care se va muta indicatorul atunci când folosim zona de paginare este gestionat prin intermediul proprietății `blockIncrement`.

Constructorul acestei componente are prototipul public `JScrollBar(int orientare, int valoare, int indicator, int minim, int maxim)`, unde orientare reprezintă orientarea dată de cele două constante enumerate mai sus, valoare reprezintă valoarea inițială (locul unde se va poziționa inițial indicatorul), minim valoarea minimă, maxim valoarea maximă, iar indicator reprezintă mărimea indicatorului.

Evenimentul pe care îl aruncă componenta JScrollBar, atunci când schimbăm poziția indicatorului, este `AdjustmentEvent`. Tratarea acestui eveniment se face cu ascultătorul `AdjustmentListener`. A se vedea exemplul care urmează.

Proprietatea `visibleAmount` oferă posibilitatea setării dimensiunii indicatorului. Dacă dimensiunile ferestrelor se modifică, trebuie ca și dimensiunile indicatorului să se modifice proporțional pentru a păstra un raport corect. În exemplul următor am suprascris metoda `paint()` a containerului de bază pentru a redesena fereastra având indicatorul componentei JScrollBar remodificat la noile dimensiuni.



```

JScrollBar derulare;
Dimension dim = this.getPreferredSize();
int i=dim.height;
// construim continut
protected void faContinut() throws Exception
{
    JPanel continut= new JPanel(); // in acest panou vom
    // adauga continutul ferestrei

```

```

this.getContentPane().add(continut, BorderLayout.CENTER);

int dis=10;
JLabel eticheta = new JLabel("Alegeti marime text",
    JLabel.CENTER);
eticheta.setAlignmentX(Component.CENTER_ALIGNMENT);

derulare = new JScrollBar( JScrollBar.VERTICAL, 10, i/10,
    10, 60);
derulare.addAdjustmentListener(new AscultatorScroll());
derulare.setUnitIncrement(1);
derulare.setBorder(BorderFactory.createEmptyBorder
    (0,0,10,0));

Font f=new Font("Tahoma", Font.BOLD, 10);
text=new JTextArea();
text.setFont(f);
text.setForeground(new Color(128,32,128));
text.setBackground(eticheta.getBackground());
text.setText("Exemplu text");
text.setEditable(false);

continut.setLayout(new BorderLayout());
continut.add(eticheta, BorderLayout.NORTH);
continut.add(text, BorderLayout.CENTER);
continut.add(derulare, BorderLayout.EAST);
}

public void paint(Graphics G)
{
    Dimension dim = this.getPreferredSize();
    int i=dim.height;
    super.paint(G);
    derulare.setVisibleAmount(i/10);
}

class AscultatorScroll implements AdjustmentListener
{
    public void adjustmentValueChanged(AdjustmentEvent e)
    {
        JScrollBar sursa = (JScrollBar)e.getSource();
        if (!sursa.getValueIsAdjusting()) {
            int fps = (int)sursa.getValue();
            Font f=new Font("Tahoma", Font.BOLD, fps);

```

```

FereastraAplicatie.text.setFont(f);
FereastraAplicatie.text.repaint();
}}}

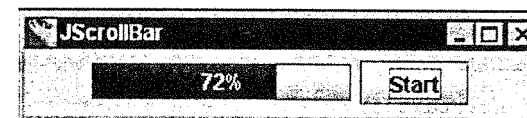
```

13.10.3. JProgressBar

Așa cum îi spune și numele, componenta `JProgressBar` este folosită pentru a indica progresul unei operații consumatoare de timp, arătând utilizatorului că operația monitorizată este activă și în desfășurare. Grafic se reprezintă printr-o bară orizontală sau verticală continuă (în cazul look-and-feel-ului Windows este formată din mici dreptunghiuri) care crește dinspre stânga spre dreapta (sau de jos în sus) de la o valoare minimă spre una maximă. Prin intermediul acesteia poate fi vizualizat procentul din acțiune care a fost deja consumat.

Culoarea fundalului și a background-ului se modifică ca în cazul oricărei componente. Proprietatea `borderPainted` specifică dacă se va desena un border în jurul componente. În timpul monitorizării unei activități vom folosi metoda `setValue()` pentru a reactualiza starea barei de progres. Această metodă trebuie folosită în firul de execuție *event dispatching thread* pentru a nu apărea probleme la desenare.

`JProgressBar` a fost concepută pentru a monitoriza operații care decurg într-un timp lung. Din punct de vedere computațional, prin timp lung se înțelege mai mult de o secundă. O atenție specială trebuie acordată logicii aplicației prin care se actualizează grafic starea barei de progres. Există două moduri de a programa o bară de progres, și anume modul analogic și modul digital. `JProgressBar` poate fi folosită pentru a arăta că ceva se petrece și nu pentru a măsura exact cât de completă este o operațiune. Acest mod de a vedea lucrurile poartă numele de mod analogic, în acest caz subliniindu-se mai mult progresele care au loc în desfășurarea operației, oferind astfel posibilitatea de a face comparații între cât s-a desfășurat din operație și ce mai este de făcut. Modul digital este folosit atunci când se dorește reprezentarea exactă a modului în care decurge operația, progresul barei de progres fiind continuu și fără salturi. În general, cele două moduri de prezentare se combină, deoarece deseori avem de a face cu operații care decurg greu, iar programatorul nu poate ști de la început când se vor termina.



```
protected void faContinut() throws Exception
```

```

{
    final int m_min=1, m_max=100;

    JPanel continut= new JPanel(); // in acest panou vom adauga
    // continutul ferestrei
    this.getContentPane().add(continut, BorderLayout.CENTER);
}

```

```

UIManager.put("ProgressBar.selectionBackground",
    Color.black);
UIManager.put("ProgressBar.selectionForeground",
    Color.white);
UIManager.put("ProgressBar.foreground", new Color
    (128,32,128));

```

```

final JProgressBar jpb = new JProgressBar();
jpb.setMinimum(m_min);
jpb.setMaximum(m_max);
jpb.setStringPainted(true);

```

```

JButton start = new JButton("Start");

```

```

class AscultatorStart implements ActionListener {
    public void actionPerformed(ActionEvent e) {

```

```

        Thread runner = new Thread() {

```

```

            int m_counter = m_min;

```

```

            public void run() {

```

```

                while (m_counter <= m_max) {

```

```

                    Runnable runme = new Runnable() {

```

```

                        public void run() {

```

```

                            jpb.setValue(m_counter);

```

```

                        }

```

```

                    };

```

```

                    SwingUtilities.invokeLater(runme);

```

```

                    m_counter++;

```

```

                }

```

```

                try {
                    Thread.sleep(100);

```

```

                }

```

```

            } catch (Exception ex) {}

```

```

        }

```

```

    };

```

```

    runner.start();

```

```

}

```

```

AscultatorStart start=new AscultatorStart();

```

```

start.addActionListener(start);

```

```

continut.add(jpb, BorderLayout.CENTER);

```

```

continut.add(start, BorderLayout.WEST);

```

```

}

```

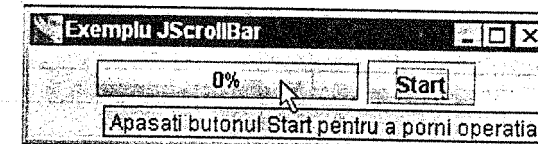
O altă metodă de a monitoriza operațiile o reprezintă folosirea clasei `ProgressMonitor` care poate fi folosită foarte eficient pentru a monitoriza operații cu fișiere având suport pentru controlul operațiilor de scriere sau citire dinspre acestea. Pentru mai multe informații sunteți invitați să consultați API-ul.

13.10.4. JToolTip

`JToolTip`-urile reprezintă mici ferestre derulante care permit asocierea de informații textuale aplicațiilor, care sunt vizualizate în momentul în care cursorul mouse-ului se află deasupra componentei. Acest mecanism este implementat în Swing astfel încât nu este nevoie să creăm componente `JToolTip`. Este îndejuns să apelăm metoda `public void setToolTipText(String text)` moștenită de fiecare componentă din clasa `JComponent`. Prin acest apel, componenta este înregistrată de un gestionar implicit care este o instanță a clasei `ToolTipManager`. De asemenea, implicit se va adăuga componentei un ascultător `MouseListener` pentru a determina când cursorul mouse-ului se va afla deasupra componentei. Pentru a anula `ToolTip`-ul asociat unei componente putem folosi un apel `setToolTipText(null)`.

Clasa `ToolTipManager` este cea care gestionează modul în care se va face prezentarea tooltip-urilor asociate componentelor. Atunci când cursorul mouse-ului intră în granițele componentei, `ToolTipManager`-ul detectează acest lucru și va aștepta 750 ms înainte de a afișa `ToolTip`-ul pentru componentă. Dacă cursorul rămâne deasupra componentei, `ToolTip`-ul va fi vizualizat timp de 4000 ms. Dacă în această perioadă de timp mutăm cursorul mouse-ului în afara componentei, tooltip-ul va dispărea. Dacă repositionăm cursorul mouse-ului deasupra componentei, `ToolTip`-ul asociat va apărea în 500 ms. Aceste trei perioade de timp pot fi setate folosind metodele puse la dispoziție de clasa de serviciu `ToolTipManager`, și anume: `setInitialDelay()`, `setDismissDelay()` și `setReshowDelay()`.

Putem obține o referință la instanța `ToolTipManager` folosind un apel `ToolTipManager toolTipManager = ToolTipManager.sharedInstance()`, putând astfel face modificări asupra managerului.



```

// ...

```

```

final JProgressBar jpb = new JProgressBar();

```

```

JButton start = new JButton("Start");

```

```

jpb.setToolTipText(

```

```

    "Apasati butonul Start pentru a porni operatia");

```

```

start.setToolTipText("Porneste");

```

```

ToolTipManager gestionar = ToolTipManager.sharedInstance();

```

```

gestionar.setInitialDelay(0);

```

Personalizarea completă a modului în care se afișează tool-tip-urile se realizează prin extinderi ale clasei `ToolTipManager`.

13.11. Ferestre interioare

Interfețele pentru documente multiple (eng. *multiple document interface* MDI) reprezintă modul standard de a gestiona și folosi mai multe documente pe aceeași suprafață de lucru, documente care sunt reprezentate de ferestre interioare. Motivul pentru care deseori se abordează această soluție a unui desktop pe care se află fișiere este faptul că dacă am avea o fereastră `JFrame` de bază care ar ocupa tot ecranul, în cazul în care ea deține focusul, va ascunde toate celelalte ferestre și dialoguri. În acest caz ar trebui să comutăm între ea și aceste ferestre, ceea ce este incomod. Din acest motiv vom considera o fereastră care va acoperi în general ecranul, iar celelalte ferestre le vom considera interioare. Prezentăm în continuare cele două clase care sunt folosite în procesul de realizare a interfețelor pentru documente multiple.

13.11.1. JDesktopPane

`JDesktopPane` este clasa care stă la baza realizării de interfețe pentru documente multiple. Putem gândi `JDesktopPane` ca un container specializat pentru ferestre interioare. Fiecare `JDesktopPane` are asociată o implementare a interfeței `DesktopManager`, care gestionează toate operațiile efectuate de ferestrele `JInternalFrame`. Dacă nu se specifică explicit un astfel de gestionar, atunci se va asocia implicit o instanță a clasei. Metodele acestui gestionar sunt automat apelate atunci când o acțiune este invocată de o fereastră `JInternalFrame` componentă (de fapt, o acțiune a utilizatorului asupra ferestrelor interioare). Câteva dintre aceste metode le vom prezenta în tabelul care urmează, însoțite de explicații :

Metoda	Explicația
<code>closeFrame(JInternalFrame f)</code>	Închide fereastra.
<code>iconifyFrame(JInternalFrame f)</code>	Iconifică fereastra.
<code>maximizeFrame(JInternalFrame f)</code>	Maximizează fereastra.
<code>restoreDownFrame(JInternalFrame f)</code>	Restorăază fereastra.
<code>openFrame(JInternalFrame f)</code>	Deschide fereastra.

Dacă dorim, spre exemplu, să minimizăm manual fereastra interioară `fereastrăInt` va trebui să realizăm un apel de forma `fereastrăInt.getDesktopPane().getDesktopManager().iconifyFrame(fereastrăInt)`. De asemenea, apelurile de metode asupra ferestrelor interioare, care se fac programatic, vor fi preluate de către gestionar pentru a fi executate. De remarcat faptul că `JDesktopPane` nu suportă borderele.

13.11.2. JInternalFrame

Ferestrele interioare se aseamănă cu ferestrele obișnuite și, chiar dacă conținutul le este gestionat asemănător prin intermediul unui panou `JRootPane`, ferestrele interioare nu sunt considerate containere de nivel înalt. Nu sunt componente „grele” neavând componente peer corespunzătoare, și din acest motiv sunt implementate independent de platformă. Ele sunt construite pentru a fi folosite în interiorul unei suprafețe de lucru (eng. *desktop*) reprezentate de clasele `JDesktopPane` din secțiunea anterioară.

Proprietățile unei ferestre interioare se pot controla mult mai ușor decât în cazul ferestrelor obișnuite. Ele se pot minimiza și maximiza și se poate specifica, de asemenea, iconul unei ferestre. Se poate specifica dacă fereastra va fi decorată cu butoane pentru a suporta redimensionarea, transformarea în icon, închiderea și maximizarea.

O fereastră interioară poate fi creată folosind un constructor de forma:

```
public InternalFrame(String titlu, //titlul ferestrei,
    boolean redim, //redimensionare
    boolean inchid, //inchidere
    boolean maxim, //maximizare
    boolean icon) ; //iconificare sau o forma
// supraincercata a acestuia.
```

Pentru a adăuga o componentă unei ferestre interioare, aceasta se adaugă panoului ce gestionează conținutul care se obține printr-un apel `getContentPane()`. Pentru vizualizarea de dialoguri se vor folosi metodele statice de genul `showInternalXXXDialog()` ale clasei `JOptionPane`. Pentru a putea fi vizualizate ferestrele interioare trebuie adăugate unui container, în principiu un `JDesktopPane`. Urmează vizualizarea ferestrei interne (în versiunile mai vechi ale platformei acestea nu erau implicit invizibile) și eventual selectarea ei pentru a deveni fereastră interioară activă.

După crearea unei ferestre interioare este nevoie să o dimensionăm, altfel ea va avea implicit dimensiunile nule și nu va fi vizibilă. Pentru aceasta se poate folosi una dintre metodele `setSize()`, `pack()` sau `setBounds()`. De asemenea, aceasta trebuie poziționată folosind `setLocation()` sau `setBounds()`, altfel va fi vizualizată în colțul din stânga sus al containerului său. Descriem mai jos o secvență de pași care ilustrează cele discutate mai sus:

```
JDesktopPane suprafataLucru= new JDesktopPane();
JInternalFrame doc = new JInternalFrame(
    "Documentul nou", true, true, true, true);
doc.setBounds(10,10,300,200);
suprafataLucru.add(doc);
doc.setVisible(true);
try { doc.setSelected(true); }
catch (Exception exe)
{ System.out.println("Exceptie"); }
this.getContentPane().add(suprafataLucru);
```

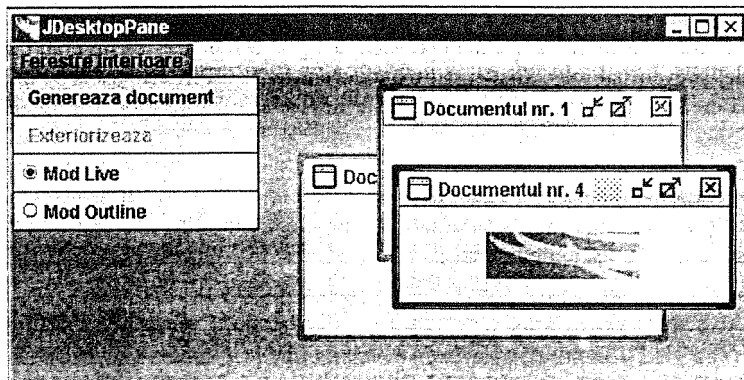
O fereastră `JInternalFrame` poate avea asociat unul sau mai mulți ascultători `InternalFrameListener` care tratează evenimentele `InternalFrameEvent` aruncate de fereastra interioară. Prezintă, în continuare, câteva dintre metodele acestui ascultător, care trebuie suprascrise pentru a trata diverse situații în care evenimentul este aruncat:

Metoda	Explicații
<code>internalFrameClosed(InternalFrameEvent e)</code>	Tratează închiderea ferestrei.
<code>internalFrameIconified(InternalFrameEvent e)</code>	Tratează iconificarea ferestrei.
<code>internalFrameOpened(InternalFrameEvent e)</code>	Tratează deschiderea ferestrei.

Pentru a nu suprascrie toate metodele avem la dispoziție o extensie a acestui gestionar, și anume `InternalFrameAdapter`. Ca optimizare, putem seta modul în care se va putea deplasa fereastra interioară cu mouse-ul prin intermediul unei apel `setDragMode()` al clasei `JDesktopPane`, având drept parametru `JDesktopPane`. `OUTLINE_DRAG_MODE` pentru deplasare optimizată sau `LIVE_DRAG_MODE` pentru deplasare standard.

Având o fereastră interioară se poate pune problema exteriorizării ei, ceea ce înseamnă transformarea ei într-o fereastră obișnuită. Acest proces se realizează în câțiva pași și constă, de fapt, în trecerea conținutului de la fereastra interioară, care va fi mai apoi închisă, spre o nouă fereastră obișnuită, proaspăt creată.

Fereastra interioară ce se dorește exteriorizată se determină, eventual, printr-un apel al metodei `getSelected()` din clasa `JdesktopPane` dacă ea este fereastra curent selectată.



```
/* Fereastra aplicatiei */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
```

```
public class FereastraAplicatie extends JFrame {
    //variabile de clasa
    static JMenuBar meniuBara;
    final Image img;
    int pozitie=1;
    JDesktopPane suprafataLucru= new JDesktopPane();
    static final Integer DOCLAYER = new Integer(5);
    int contor=0;

    //constructorul
    public FereastraAplicatie() throws Exception {
        super("JDesktopPane");
        // positionam fereastra si-i setam dimensiunile
        Toolkit t=this.getToolkit();
        Dimension marimeEcran = Toolkit.getDefaultToolkit().
            getScreenSize();
        setBounds ( pozitie, pozitie, marimeEcran.width-
            pozitie, marimeEcran.height-20*pozitie);
        // setam iconul din coltul din stanga sus al ferestrei
        img=t.getImage("icon.jpg");
        this.setIconImage(img);
        this.getContentPane().setLayout(new BorderLayout());
        // construim partile componente
        faMeniu();
        faContinut();
        // specificam modul de reactie al ferestrei la inchidere
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                iesire();
            }
        });
    }

    protected void faMeniu() {
        meniuBara = new JMenuBar(); // cream bara de meniu
        meniuBara.setOpaque(true);
        JMenu optiuni= faMeniuOptiuni(); //cream un meniu
        optiuni.setMnemonic('M'); // definim shortcut (Alt+M)
        meniuBara.add(optiuni); // adaugam meniul optiuni la
        bara de meniu
        setJMenuBar(meniuBara);
    }
}
```



```

// construim meniul optiuni
protected JMenu faMeniuOptiuni() {
    JMenu unelte= new JMenu("Ferestre interioare");
    JMenuItem gen=new JMenuItem("Genereaza document");
    final JMenuItem ext=new JMenuItem("Exteriorizeaza");
    ext.setEnabled(false);

    gen.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JInternalFrame doc = new JInternalFrame("Documentul nr."
                +(++contor),
                true, //redimensionabila
                true, //se poate inchide
                true, //se poate minimiza
                true);
            doc.getContentPane().add(new JLabel(new ImageIcon(img)));

            doc.addInternalFrameListener(new InternalFrameAdapter() {
                public void internalFrameClosed(InternalFrameEvent e) {
                    {
                        ext.setEnabled(suprafataLucru.
                            getSelectedFrame() !=null? true:false);
                    }
                }
            });

            doc.setBounds(10*contor,10*contor,300,200);
            suprafataLucru.add(doc);
            doc.setVisible(true);
            try {
                doc.setSelected(true);
                ext.setEnabled(suprafataLucru.getSelectedFrame() !=null?
                    true:false);
            }
            catch(Exception exe)
            { System.out.println("Exceptie");}
            repaint();
        }
    });

    unelte.add(gen);
    unelte.addSeparator();

    unelte.add(ext);
    unelte.addSeparator();

```

```

ext.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JInternalFrame ferAct=suprafataLucru.getSelectedFrame();
        JFrame ferNoua=new JFrame();
        if (ferAct!=null)
        {
            ferAct.setVisible(false);
            ferNoua.setContentPane(ferAct.getContentPane());
        }
        ferAct.disable();
        ext.setEnabled(false);
        ferNoua.setBounds(10*contor,10*contor,290,190);
        ferNoua.setVisible(true);
    }
});

JRadioButtonMenuItem o1 =
    new JRadioButtonMenuItem("Mod Live");
o1.setSelected(true);
JRadioButtonMenuItem o2 =
    new JRadioButtonMenuItem("Mod Outline");
// adaugam ascultatorul pentru optiunea1
ButtonGroup grup=new ButtonGroup();
grup.add(o1);
grup.add(o2);

o1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        suprafataLucru.setDragMode(
            JDesktopPane.LIVE_DRAG_MODE);
    }
});
// adaugam ascultatorul pentru optiunea2
o2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        suprafataLucru.setDragMode(
            JDesktopPane.OUTLINE_DRAG_MODE);
    }
});
// adaugam optiunile la meniul
untele.add(o1);
untele.addSeparator();
untele.add(o2);
return unelte;
}

// construim continut
protected void faContinut() throws Exception

```

```

    {
        this.getContentPane().add(suprafataLucru);
    }

    public void iesire() {
        System.exit(0); // parasim aplicatia
    }
}

```

13.12. Recomandări și optimizări

Ca o primă observație, se recomandă despărțirea părții de logică a aplicației (eng. *business logic*) de interfața grafică din motive de reutilizare a codului, atât pentru interfață, cât și pentru partea de logică a aplicației. Astfel, se poate interveni în oricare parte a aplicației, fără a o afecta pe cealaltă, iar partea logică poate fi folosită și prin intermediul altor interfețe grafice (aparținând altor aplicații). Acest lucru poate fi benefic în cazul aplicațiilor distribuite în care partea de logică a aplicației se poate găsi pe o altă mașină, aflată la distanță. Despărțirea se face prin construirea a două clase diferite, una pentru logică și alta pentru interfață, interfața realizând interogări asupra părții logice a aplicației ca urmare a interacțiunii cu utilizatorii.

De asemenea, folosirea de denumiri sugestive pentru numele componentelor reprezintă un lucru benefic, care se va observa pe parcursul dezvoltării aplicației, chiar dacă pare mai simplu să folosim în grabă nume scurte și fără înțeles. Spre exemplu, dacă dăm nume sugestive mai multor componente folosind expresii regulate, prin intermediul Reflection API putem obține foarte ușor referințe despre acele componente. De asemenea, este indicată folosirea de comentarii care să ne ajute mai apoi în descoperirea greșelilor. O scriere elegantă a codului aduce beneficii în procesul de depanare și de menținere a aplicației.

Cel mai important motiv pentru care interfețele grafice merg greoi îl reprezintă nefolosirea eficientă a firelor de execuție. Chiar dacă în Swing și desenarea, și expedierea evenimentelor se realizează într-un singur fir de execuție, pentru optimizarea codului trebuie exploatate la maximum firele de execuție și metodele `invokeAndWait()`, respectiv `invokeLater()` (a se vedea secțiunea dedicată firelor de execuție în Swing).

Un alt motiv de încetinire în execuția aplicației îl reprezintă crearea componentelor prin constructori care nu folosesc modele explicite, ci modele implicite. Adăugarea de elemente direct componentelor prin metodele pe care acestea le pun la dispoziție determină aruncarea de evenimente inutile, care îngreunează aplicația. De aceea, este recomandată crearea de modele proprii care vor fi folosite mai apoi în procesul de creare a componentelor. Înțelegerea arhitecturii MVC poate îmbunătăți considerabil timpul aplicației. Pentru aceasta trebuie să parcurgeți partea introductivă a acestui capitol.

Prezentăm în continuare optimizări ale pachetului Swing preconizate în viitoarele distribuții Java.

Versiunea 1.4 a distribuției Java permite folosirea directă a acceleratorilor grafici hardware pentru PC-urile moderne.

Versiunea 1.4.2 a platformei Java 2 își propune să aducă următoarele îmbunătățiri pachetului Swing:

- îmbunătățirea vitezei de startare a aplicațiilor și appleturilor (inclusiv a celor cu interfețe grafice Swing);
- reducerea consumului de memorie și creșterea performanțelor;
- un nou pachet pentru API-ul `FileChooser`, din cauza faptului că actualul API funcționează defectuos pentru cantități mari de date;
- stabilizarea opțiunii `setDisplayMode()` care permite folosirea modului full screen, aceasta fiind prost implementată în versiunile anterioare;
- look-and-feel-ul XP și look-and-feel-ul GTK vor veni împreună cu noua distribuție.

Versiunea 1.5 a platformei Java va aduce:

- simplificarea API-ului Swing. Spre exemplu, se presupune că anumite componente, precum `JFrame` și `JDialog`, nu vor mai avea un mod atât de complicat de gestiune a conținutului, care presupune studierea fără nici un folos a altor componente, precum `JRootPane`, `JGlassPane` etc.;
- îmbunătățirea lucrului cu ferestre interioare;
- îmbunătățiri în modul de lucru anti-aliased.

13.13. Concluzii

Acest capitol prezintă librăria Swing, care pune la dispoziția dezvoltatorilor de aplicații Java componentele necesare pentru realizarea interfețelor grafice.

Capitolul debutează cu o introducere în JFC și cu o descriere a componentelor Swing și a câtorva aspecte fundamentale legate de acestea. De asemenea, sunt prezentate toate pachetele care alcătuiesc librăria Swing.

Urmează principii care stau la baza librăriei Swing, cum ar fi modelul de structurare MVC, modelul `UIDelegate`, Look-and-Feel-ul aplicațiilor, prezentarea librăriei Swing în raport cu AWT. În cazul look-and-feel-ului, am precizat referințe spre pagini de unde cititorii pot să le descarce.

Subcapitolul „Fundamentele Swing” prezintă detaliat sistemul de evenimente și ascultători, firele de execuție în Swing, desenarea componentelor, concordanța cu JavaBeans, respectiv gestiunea focusului. Capitolul se încheie cu o clasificare a componentelor care determină structura subcapitolelor ce urmează, în care componentele vor fi prezentate detaliat, grupate după cum se va vedea în continuare.

Secțiunea „Containerele de bază” prezintă ferestrele Swing și clasele și interfețele care au legătură cu acestea.

Secțiunea „Componentele atomice simple” prezintă componentele etichete, butone, borduri, liste.

Subcapitolul „Componentele atomice complexe” prezintă componentele text, tabele și arbori.

Urmează un subcapitol dedicat meniurilor și barelor de unelte.

Un subcapitol dedicat dialogurilor.

Un subcapitol dedicat componentelor pentru gestiunea progresului și pentru derulare.

Un subcapitol destinat ferestrelor interioare.

Ultima secțiune cuprinde recomandări pentru dezvoltatori și optimizări anunțate pentru viitoarele distribuții ale platformei Java.

13.14. Test grilă

Întrebarea 13.14.1. Ce reprezintă Swing?

- a) ☐ O interfață prietenoasă cu utilizatorii.
- b) ☐ O librărie de componente grafice.
- c) ☐ O componentă grafică.
- d) ☐ O metodă de a realiza efecte de animație.

Întrebarea 13.14.2. Care este deosebirea între Swing și AWT?

- a) Componentele Swing sunt scrise totalmente în Java, pe când în AWT, componentele sunt scrise folosind cod nativ.
- b) Componentele AWT au vizualizarea dependentă de sistemul de operare, iar în Swing componentele pot avea o aceeași vizualizare, indiferent de sistemul de operare.
- c) Componentele Swing sunt mai ușoare decât componentele AWT.

Întrebarea 13.14.3. Care dintre afirmațiile următoare sunt adevărate?

- a) Fiecare componentă AWT are o componentă corespondentă în Swing.
- b) Componentele Swing au denumirea componentelor AWT, la care se adaugă prefixul „J”.
- c) O componentă reprezintă un obiect cu reprezentare grafică.

Întrebarea 13.14.4. Cum se realizează vizualizarea unei componente grafice?

- a) Vizualizarea componentelor grafice se realizează o dată cu apelarea constructorului.
- b) Vizualizarea componentelor grafice se realizează prin apelarea metodei `setVisible()`.
- c) Vizualizarea componentelor grafice se realizează prin adăugarea componentei la un container deja vizibil.
- d) Vizualizarea componentelor grafice se realizează printr-un apel `setContainerLayout(null)`.

Întrebarea 13.14.5. Ce efect va avea pentru un container de bază apelul metodei `setBounds(10,10,100,200)`?

- a) Componenta va avea acum lățimea 90 și înălțimea 190.

- b) Componenta se va poziționa la punctul având coordonatele colțului din stânga sus (10, 10) și coordonatele colțului din dreapta jos (100, 200).
- c) Componenta va avea lățimea 100 și înălțimea 200.

Întrebarea 13.14.6. Ce reprezintă pluggable look-and-feel-ul pentru o componentă grafică?

- a) Posibilitatea de a ascunde componenta în momentul rulării interfeței.
- b) O metodă de a schimba aspectul componentei și al modului în care aceasta interacționează cu utilizatorii.
- c) O metodă standard de a adăuga plugg-in-uri aplicațiilor Java.

Întrebarea 13.14.7. Care dintre afirmațiile următoare sunt corecte?

- a) Un model de date păstrează datele unei componente grafice.
- b) Toate componentele Swing au modele.
- c) Modelele dețin informații despre vizualizarea tabeli.
- d) Modelele interacționează cu vizualizarea doar prin intermediul evenimentelor.

Întrebarea 13.14.8. Fie secvența de cod:

```

JButton b=new JButton();
public class TratareIesire implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        dispose(); // inchide fereastra curenta
    }
}
ActionListener[] al=(ActionListener[]) (b.getListeners(Action
Listener.class));
if(al.length==0)
    b.addActionListener(new TratareIesire());

```

Ce reprezintă secvența de cod precedentă?

- a) Adăugarea mai multor ascultători unui buton.
- b) Adăugarea unui ascultător unui buton numai în cazul în care acesta nu mai are un altul asociat.
- c) Ștergerea unui ascultător.

Întrebarea 13.14.9. Care dintre afirmațiile următoare sunt adevărate?

- a) Unei componente îi putem adăuga mai mulți ascultători de același tip.
- b) Un adaptor (eng. *adapter*) este o clasă abstractă care extinde o interfață și pe care o moștenim pentru a nu fi nevoiți să implementăm fiecare metodă a interfeței respective.
- c) `KeyEvent` este evenimentul generat de apăsarea unui buton al mouse-ului.
- d) `MouseListener` este ascultătorul care permite captarea evenimentelor generate de mouse.

Întrebarea 13.14.10. Care dintre afirmațiile următoare sunt adevărate?

- a) Swing nu permite folosirea altor fire de execuție diferite de *event-despatching thread*.
- b) Dacă un container este redesenat, toate componentele conținute sunt redesenate și ele.
- c) Metoda *double-buffering* dă posibilitatea desenării direct pe monitor a componentelor.
- d) Swing permite folosirea a numai patru fonturi.
- e) Focusul reprezintă fluxul acces spre dispozitivul standard de intrare.

Întrebarea 13.14.11. Care dintre afirmațiile următoare sunt adevărate?

- a) JPanel este un container de bază;
- b) JRootPane este container intermediar;
- c) JApplet reprezintă un applet.

Întrebarea 13.14.12. Cum se adaugă un buton unui container de bază JFrame?

- a)

```
JFrame f=new JFrame();
JPanel continut = (JPanel)f.getContentPane();
continut.add(new JButton("OK"));
```
- b)

```
JFrame f=new JFrame();
f.getContentPane().add(new JButton("OK"));
```
- c)

```
JFrame f=new JFrame();
f.add(new JButton("OK"));
```

Întrebarea 13.14.13. Care dintre afirmațiile următoare sunt adevărate?

- a) Panoul ContentPane este gestionat implicit de FlowLayout.
- b) Orice alt panou diferit de contentPane este gestionat implicit de FlowLayout.
- c) Apelul `setLayout(null)` determină ca în containerul curent poziționarea să se facă absolut (relativ la colțul din stânga sus al ecranului).

Întrebarea 13.14.14. Cum se face ca un arbore JTree să poată fi derulat de componenta JScrollPane?

- a)

```
JScrollPane sp= new JScrollPane(new JTree(model));
```
- b)

```
JScrollPane sp= new JScrollPane();
sp.add(new JTree(model));
```
- c)

```
JScrollPane sp= new JScrollPane();
sp.setViewportViewView(new JTree(model));
```

Întrebarea 13.14.15. Care dintre afirmațiile următoare sunt adevărate?

- a) JTabbedPane permite crearea unui panou format din alte două subpanouri alăturate.
- b) Panourile care alcătuiesc un JSplitPane respectă dimensiunile minime ale componentelor.
- c) JTabbedPane reprezintă o stivă de componente dintre care numai una este vizibilă la un moment dat.

Întrebarea 13.14.16. Cum se face ca o tabelă JTable să poată fi derulată de componenta JScrollPane?

- a)

```
JScrollPane sp = new JScrollPane(new JTable(model));
```
- b)

```
JScrollPane sp = new JScrollPane();
sp.add(new JTable(model));
```
- c)

```
JScrollPane sp = new JScrollPane();
sp.add(new JTable(model));
sp.setScrollable(true);
```

Întrebarea 13.14.17. Fie secvența de cod:

```
JPanel p = new JPanel();
JLabel e = new JLabel("Eticheta:");
e.setDisplayedMnemonic('E');
p.add(et, BorderLayout.EAST);
JTextField t = new JTextField(7);
e.setLabelFor(t);
p.add(t, BorderLayout.WEST);
```

Ce se va întâmpla prin apăsarea combinației *Alt+E*?

- a) Eticheta e va deține focusul.
- b) Câmpul t va deține focusul.
- c) Nici o componentă nu va mai deține focusul.

Întrebarea 13.14.18. Care dintre ascultătorii următori indică bifarea unui JCheckBox?

- a) ActionListener
- b) EventListener
- c) ItemListener
- d) MouseListener

Întrebarea 13.14.19. Care dintre afirmațiile următoare sunt adevărate?

- a) Un obiect bordură (eng. *border*) este o componentă grafică.
- b) O bordură `TitledBorder` permite asocierea unui titlu unui grup de componente simple.
- c) Un tooltip reprezintă o bară de unelte.

Întrebarea 13.14.20. Care dintre afirmațiile următoare sunt adevărate?

- a) La un moment dat, într-o listă poate fi selectat un singur element.
- b) JComboBox reprezintă o listă expandabilă.
- c) Formatarea conținutului pentru JList și JComboBox se face prin intermediul clasei `ListCellRenderer`.
- d) Componenta JSpinner are un model de date numit `JSpinnerListModel`, care permite setarea facilă a datei calendaristice.

Întrebarea 13.14.21. Care dintre următoarele afirmații este adevărată?

- a) Secvența de cod care urmează va arunca o excepție:

```
JPasswordField p= new JPasswordField();
```

```
String text=p.getText();
```

- b) Metodele `copy()` și `cut()` folosesc clipboard-ul sistemului de operare și, de aceea, printr-o operațiune `paste()` executată dintr-o altă aplicație, eventual nu Java, va alipi conținutul.
- c) Pentru a face ca o componentă text să nu poată obține focusul într-un ciclu focus, se va apela metoda `transferFocus()`.
- d) Modelul `PlainDocument` ne dă prin suprascriere posibilitatea de a personaliza componentele text.
- e) Swing nu oferă suport pentru operațiile `undo/redo`.

Întrebarea 13.14.22. Care dintre afirmațiile următoare sunt adevărate?

- a) Întotdeauna o componentă `JTable` are asociat un model de date, chiar dacă nu specificăm explicit acest lucru.
- b) Prin modificări efectuate asupra datelor din model, se va actualiza automat și vizualizarea.
- c) Pentru a crea o tabelă folosind modelul `AbstractTableModel`, este îndeajuns să suprascriem metodele `getRowCount()` și `getColumnCount()`.
- d) Modelul de date `AbstractTableModel` ține datele întotdeauna într-un `Vector` având elemente `Vector`.
- e) O tabelă permite derularea conținutului fără a trebui inclusă într-un panou `JScrollPane`.

Întrebarea 13.14.23. Fie secvența de cod:

```
class Copie extends AbstractAction
{
    Copie(String nume, ImageIcon icon)
    { super(nume, icon); }
    public void actionPerformed(ActionEvent e)
    { text.copy(); }
}

JMenuBar m=new JMenuBar();
JMenu e = new JMenu("Editare");
JMenuItem c=e.add(new Copie("Copie",new ImageIcon
    ("copie.gif")));
m.add(e);
```

Ce se va întâmpla la execuția codului?

- a) Se va arunca o excepție care marchează faptul că nu se poate adăuga un eveniment unui meniu.
- b) Se va crea un meniu bară, cu un meniu *Editează*, având o opțiune *Copie*.
- c) Se va arunca o excepție, pentru că opțiunile meniurilor Java nu pot avea iconuri.

Întrebarea 13.14.24. Care dintre afirmațiile următoare sunt adevărate?

- a) Clasa `JOptionPane` permite crearea de dialoguri numai folosită împreună cu clasa `JDialog`.

b) Fie secvența de cod:

```
Object[] optiuni = {"Da!", "Nu!"};
int nr = JOptionPane.showOptionDialog(fereastra, "Alegeți opțiune",
    "Mesaj cu opțiuni personalizate", JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE, null, optiuni, optiuni[0]);
```

Dacă vom apăsa butonul „Da!”, nr va lua valoarea 1.

- c) În cazul în care folosim componenta `JFileChooser`, prin alegerea unui nume de fișier, în aplicație vom obține o referință `File` spre acesta.
- d) Fereastra `JColorChooser` blochează întotdeauna celelalte ferestre ale aplicației.

Întrebarea 13.14.25. Care dintre afirmațiile următoare sunt adevărate?

- a) `DefaultBoundedRangeModel` reprezintă modelul de date pentru clasele `JSlider`, `JScrollBar`, `JProgressBar`.
- b) `JScrollBar` este folosită de componenta `JScrollPane` pentru a derula conținutul.
- c) În cazul componentei `JProgressBar`, modificarea poziției indicatorului se realizează prin apeluri `moveTo()`.

Întrebarea 13.14.26. Ce determină secvența de cod?

```
JDesktopPane suprafataLucru= new JDesktopPane();
suprafataLucru.setDragMode(JDesktopPane.OUTLINE_DRAG_MODE);
```

- a) Ferestrele interioare conținute în panoul `suprafataLucru` nu vor mai putea fi mutate.
- b) Ferestrele interioare conținute în panoul `suprafataLucru` vor putea fi exteriorizate.
- c) Ferestrele interioare conținute în panoul `suprafataLucru` vor putea fi mutate într-un mod mai puțin consumator de resurse.

13.15. Exerciții propuse spre implementare

Exercițiul 13.15.1. Realizați animații folosind API-ul Java2D.

Exercițiul 13.15.2. Testați modul în care se vor comporta componentele `JTable` și `JTree` în cazul unor volume mari de date. Gândiți-vă și încercați diverse optimizări.

Exercițiul 13.15.3. Realizați o aplicație care monitorizează toți ascultătorii din mașina Java curentă și evenimentele pe care aceștia le ascultă.

Exercițiul 13.15.4. Testați efectele pe care le au firele de execuție utilizator care modifică interfața Swing pentru o aplicație oarecare. Remediați deficiențele folosind metodele indicate în secțiunea dedicată firelor de execuție.

13.16. Proiecte propuse spre implementare

Proiectul 13.16.1. Realizați un editor vizual pentru documentele de tip XML, folosind componenta `JTable`. Se va evita folosirea componentelor `JTree` și a componentelor text.

Proiectul 13.16.2. Realizați un convertor XSL având interfață grafică prietenoasă. Pentru a permite editarea se va folosi clasa `JEditorPane`, despre care găsiți informații în API-ul Java.

Proiectul 13.16.3. Creați un look-and-feel pentru Java, eventual pornind de la un look-and-feel deja existent.

Proiectul 13.16.4. Creați un administrator al resurselor mașinii virtuale Java. Acesta va avea o interfață grafică atractivă și va permite curățarea memoriei apelând *garbage collector*, va arăta consumul de memorie etc.

Proiectul 13.16.5. Realizați un convertor care transformă aplicațiile grafice ce folosesc AWT în aplicații grafice folosind Swing. În tutorialul Java există explicații legate de o astfel de conversie.

Proiectul 13.16.6. Realizați un mediu vizual pentru Java respectând convențiile din specificația JavaBeans. Acesta va avea o structură modulară, permițând un nucleu de funcționalități la care se vor putea adăuga în timp și altele.

Proiectul 13.16.7. Realizați un browser Web folosind Swing. Acesta va putea vizualiza și sursele paginilor vizitate.

Proiectul 13.16.8. Realizați un editor HTML folosind principiile MDI (multiple document interface). Pentru mai multe informații, a se vedea secțiunea „Ferestre interioare”.

Proiectul 13.16.9. Realizați un gestionar de fișiere (echivalentul aplicației Windows Commander pentru Java).

Proiectul 13.16.10. Realizați o aplicație care să fie un editor și browser Wap. Specificațiile Wap sunt disponibile pe Internet.

14. Sistemul Help pentru Java

14.1. Cuvinte cheie

- JavaHelp
- HelpSet
- help senzitiv la context

14.2. Convenții

- Vom folosi denumirea generică de *help* în sens de ajutor, asistență, netraducând-o în limba română. În principiu, înseamnă modalitatea prin care dezvoltatorul produsului informatic explică, prezintă utilizatorilor aplicația. Poate fi folosit și cu sensul mai larg de documentație, caz în care nu ne referim neapărat la o aplicație.
- Prin *sistem help* înțelegem modalitatea pe care un limbaj o oferă programatorilor pentru a atașa help aplicațiilor. Sistemul help pentru platforma JAVA poartă numele *JavaHelp*.
- Prin *HelpSet* vom înțelege totalitatea fișierelor care participă la realizarea sistemului de help pentru o aplicație oarecare dezvoltată în limbajul Java, păstrând astfel denumirea întâlnită în specificație și în ghidurile de utilizare.

14.3. Principii de bază

14.3.1. Introducere

Aplicațiile complexe necesită și un mod de prezentare a documentației aferente, deoarece este vitală înțelegerea de către utilizatori a modului în care pot interacționa cu aplicația pentru a-și atinge obiectivele. Java pune la dispoziția dezvoltatorilor un mod standard și ușor de folosit pentru realizarea documentației aplicațiilor pe care aceștia le creează. Sistemul help pentru Java este reglementat printr-o specificație gestionată de firma SUN Microsystems, de care trebuie să țină seama toți cei care dezvoltă sisteme help pentru acest limbaj. De asemenea, firma SUN pune la dispoziția programatorilor și o implementare realizată totalmente în Java a acestei specificații, deci un sistem help efektiv, numit JavaHelp, care face subiectul acestui capitol. Această implementare oferă o interfață simplă, prin care dezvoltatorii pot adăuga help aplicațiilor, și permite chiar personalizarea sistemului help pentru a se potrivi cu stilul adoptat în realizarea produsului informatic. JavaHelp permite adăugarea de help atât aplicațiilor standalone, cât și appleturilor sau componentelor JavaBeans. De asemenea, prin intermediul unui server de help se poate adăuga asistență și aplicațiilor scrise în alte limbaje decât Java.

Folosind JavaHelp, autorul documentației este responsabil pentru scrierea temelor (eng. *topics*) care tratează anumite subiecte într-un mod standard, și anume în format HTML, precum și a unei serii de metadata (informații despre date) structurate în fișiere XML. API-ul JavaHelp funcționează ca un vizualizator care prezintă utilizatorului într-un mod standard (eventual personalizat) documentația alcătuită din multitudinea de teme, ținând cont de modul în care acestea sunt structurate prin intermediul metadatelor. Putem spune că partea de structurare este despărțită de conținut și acestea sunt despărțite de modul vizualizare care este dat de API-ul

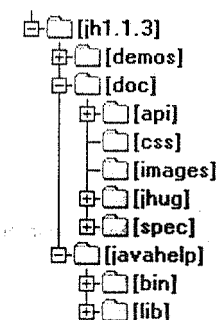
JavaHelp. Astfel, cel care realizează documentația nu trebuie să știe cum se parsează XML sau cum se folosește HTML în Java, ci se concentrează numai pe scrierea documentației și pe atașarea ei aplicației. Fișierele HTML împreună cu fișierele XML (chiar dacă uneori nu au extensia .xml) poartă numele de HelpSet, la fel ca și clasa de bază a API-ului, cea care face legătura dintre aplicație și documentație.

Pentru exemplificarea noțiunilor prezentate în acest capitol vom folosi drept referință aplicația JAIS care gestionează colecțiile și fondurile aflate în păstrare la DJAN IAȘI.

14.3.2. Prezentarea pachetului JavaHelp

Acest capitol se bazează pe implementarea implicită a sistemului help pentru Java, oferit de firma SUN, și anume JavaHelp, care reprezintă un pachet opțional pentru JDK1.1 și J2SDK. Acesta funcționează pe ambele platforme, cu unele optimizări pentru a doua, optimizări posibile datorită dezvoltării și maturizării limbajului Java.

Ultima distribuție a sistemului JavaHelp, și anume *JavaHelp1.1.3* o puteți descărca gratuit de la adresa: <http://java.sun.com/products/javahelp/>. După dezarhivarea pachetului se obține următoarea structură de directoare:



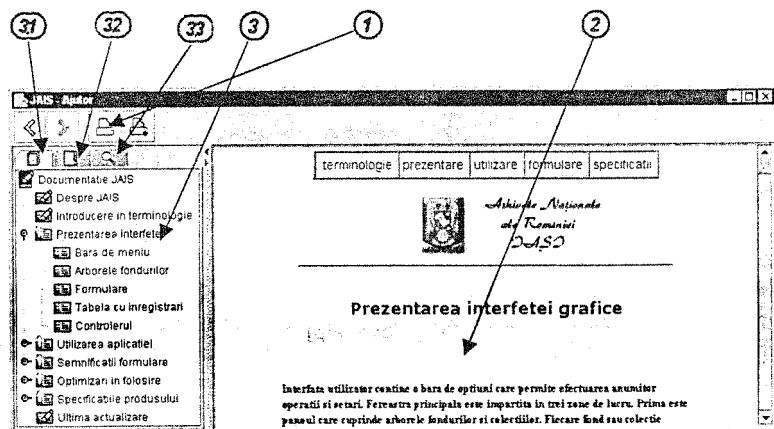
Pachetul *JavaHelp1.1.3* vine și cu documentația aferentă, care se găsește în directorul /doc al distribuției. Aceasta este alcătuită din javadoc-ul care prezintă API-ul în forma standard bine cunoscută, specificația cu versiunea 1.0 a sistemului help pentru Java și un ghid de utilizare pentru această distribuție poziționat în directorul /doc/jhug. Ghidul (eng. *JavaHelp User Guide*) este prezentat chiar sub forma de JavaHelp, deci partea de HTML care stă la baza lui se poate vizualiza cu un browser Web oarecare. Același ghid de utilizare îl regăsim și în format PDF în chiar rădăcina pachetului. De asemenea, pachetul vine și cu o serie de exemple concrete de utilizare care se află în directorul /demos.

În directorul javahelp/lib se găsesc mai multe pachete care pot fi incluse în CLASSPATH-ul aplicației, fiecare dintre acestea având un anumit specific. Clasele sistemului JavaHelp sunt distribuite în mai multe fișiere JAR, utilizatorul incluzând în CLASSPATH arhiva corespunzătoare nevoilor aplicației pe care o dezvoltă:

Arhiva	Conținut
jh.jar	Reprezintă librăria standard și include tot ceea ce este necesar pentru vizualizarea help-ului folosind modurile standard de navigare (cuprins, index, căutare).
jhbasic.jar	Un subset al jh.jar care nu cuprinde suport pentru căutări. Acesta poate fi folosit pentru sisteme simple de help care nu necesită căutări sau în cazul sistemelor în care spațiul e important.
jhall.jar	Include toate clasele sistemului JavaHelp, inclusiv unele necesare pentru crearea bazei de date (indexului) necesare căutărilor.
jsearch.jar	Motorul de căutare implicit utilizat în sistemul JavaHelp.

Directorul `java\help\bin` cuprinde o serie de unelte standard necesare în procesul de dezvoltare a documentației, și anume `jhindexer`, care reprezintă motorul de indexare, și `jhssearch`, reprezentând motorul de căutare.

14.3.3. Alcătuirea ferestrei standard JavaHelp



Interfața grafică standard a sistemului JavaHelp constă dintr-o bară de unelte și două panouri:

1. **Bara de unelte** – permite navigarea înainte și înapoi între temele documentației la fel ca într-un browser obișnuit, precum și listarea la imprimantă a documentației.
2. **Panoul conținut** – prezintă help-ul propriu-zis în formatul HTML 3.2 plus unele elemente Java cum ar fi ferestre secundare și ferestre pop-up. Conținutul este prezentat sub formă de teme, în fapt fișiere HTML care descriu un subiect ales în prealabil din panoul de navigare de către utilizatorul aplicației.
3. **Panoul de navigare** – o interfață tabbed care permite utilizatorilor să comute între cuprins, index și căutare, acestea reprezentând metodele standard puse la dispoziție pentru navigarea în documentație, prezentate în continuare:

- 3.1. **Cuprinsul** – este o reprezentare arborescentă a temelor constitutive ale sistemului help atașat aplicației, fiind de fapt cuprinsul documentației. Suportă niveluri nelimitate și alăturarea mai multor cuprinsuri corespunzătoare diferitelor HelpSet-uri. De fapt, această structură arborescentă este reprezentarea vizuală, folosind `JTree`, a unui document XML care codifică conținutul documentației. Vizualizarea cuprinsului este sincronizată cu panoul conținut, numele temei vizualizate în conținut fiind marcat în cuprins.
- 3.2. **Indexul** – tot o structură arborescentă, care reprezintă de această dată indexul documentației. Suportă alăturarea mai multor indexuri. Ca și în cazul cuprinsului, este vorba despre înfășurarea unui fișier XML folosind `JTree`. La fel există sincronizare între panoul ce conține indexul și panoul conținut.
- 3.3. **Căutarea** – un motor flexibil de căutare poate fi folosit într-o diversitate de medii de rețea. Numele temelor returnate în urma căutărilor sunt prezentate în mod descendent, în funcție de potrivire.

14.3.4. Principii pe care sistemul JavaHelp le respectă

- **Compresia și încapsularea** – formatul standard JAR poate fi folosit pentru a încapsula informațiile Help într-un singur fișier comprimat. Sistemul JavaHelp lucrează în mod egal cu informații help care nu sunt comprimate în fișiere JAR, această facilități permițând dezvoltatorilor să aibă acces la fișiere în timpul creării help-ului fără să piardă timp pentru a comprima fișierele componente.
- **Scufundarea ferestrelor Help** – ferestrele Help (individuale sau combinate) pot fi scufundate direct în interfețele grafice ale aplicațiilor.
- **Help senzitiv la context** – help-ul poate fi activat din interiorul aplicațiilor folosind diverse mecanisme cum ar fi apăsarea tastei F1.
- **Personalizare** – sistemul JavaHelp este proiectat pentru a permite o mare flexibilitate atât în ceea ce privește interfața utilizator, cât și în ceea ce privește funcționalitatea.
- **Combinări** – informații help din diferite surse pot fi combinate și prezentate utilizatorului final.
- **Suport pentru JavaBeans** – API-ul JavaHelp permite componentelor JavaBeans să specifice informații help care pot fi prezentate utilizatorilor finali (eventual, combinate cu informații suplimentare).

14.4. Dezvoltarea efectivă a help-ului

Reamintim că HelpSet-ul reprezintă totalitatea fișierelor necesare creării sistemului JavaHelp pentru aplicația curentă. Prezentăm, în continuare, lista pașilor necesari a fi parcurși pentru crearea HelpSet-ului:

1. Crearea temelor HTML (în general, în fișiere separate) care formează împreună conținutul documentației.
2. Crearea fișierului HelpSet care leagă între ele toate metadatele.
3. Crearea fișierului Map care asociază identificatori temelor HTML.

4. Crearea fișierului cuprins care descrie conținutul documentației.
5. Crearea fișierului index care descrie indexul documentației.
6. Crearea bazei de date necesare mai apoi în procesul de căutare.
7. Compresia și încapsularea tuturor fișierelor într-o arhivă JAR.

O dată HelpSet-ul creat, pe parcursul procesului de dezvoltare puteți să-l vizualizați, fără să-l atașați aplicației, folosind o unealtă care vine o dată cu pachetul JavaHelp și care este localizată în directorul `demo/bin/`, și anume `hsviewer`. Sintaxa comenzii de apelare este `java -jar hsviewer.jar [-helpset nume_HS]`, unde `nume_HS` reprezintă fișierul HelpSet (cel cu extensia `.hs`). Dacă nu precizăm nici un fișier de intrare, îl veți putea alege din interfața grafică care va apărea.

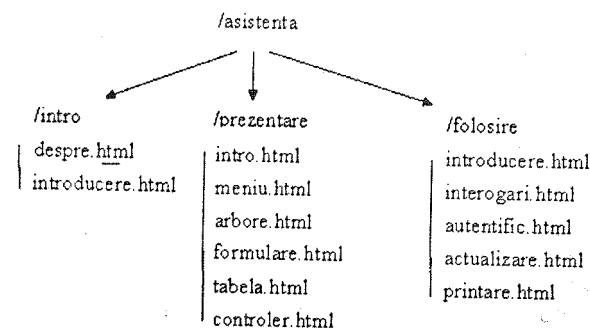
Pentru a realiza toate operațiile de la pașii precedenți, puteți folosi diverse unelte (eng. *tools*) puse la dispoziție gratuit sau contra cost de diverse firme de soft. O listă a acestora o puteți găsi în pagina de bază a produsului JavaHelp <http://java.sun.com/products/javahelp/industry.html>.

14.4.1. Temele HTML

Primul pas constă în scrierea conținutului efectiv al documentației, care este alcătuit dintr-o mulțime de fișiere HTML, în general, fiecare dintre ele descriind o anumită temă. De exemplu, pentru o aplicație, o temă poate descrie o funcționalitate sau un element al interfeței grafice. Între aceste fișiere HTML pot exista legături (eng. *links*) și se recomandă folosirea lor în forma relativă, adică având drept referință directorul curent. Spre exemplu, link-ul relativ de forma `<a href=".. /prezentare/ meniu.html">Prezentarea meniului</a>` este de preferat link-ului absolut de forma `<a href="C:/aplicatia/asistenta/prezentare/meniu.html">Prezentarea meniului</a>`, deoarece link-urile relative rămân valide și atunci când temele se arhivează sau se mută pe mașina utilizatorului. Separatorul între componentele URL-urilor trebuie să fie simbolul `„/”`.

Panoul din dreapta care vizualizează conținutul se bazează pe componenta `Swing JEditorPane`. Din această cauză pot apărea probleme dintre care unele au fost fixate în timp sau pot fi evitate într-un fel sau altul (a se vedea ghidul de utilizare pentru amănunte). Este recomandată folosirea strictă a tag-urilor HTML 3.2.

Din experiență, se recomandă să fie cât mai multe fișiere (fiecare temă să fie într-un fișier HTML separat), și nu fișiere mari în care să se folosească ancure cu nume (tagul `<a>`), din cauza modului greoi în care se face poziționarea pe ancure. Procesul de deschidere a unui fișier mic este mai rapid decât poziționarea pe o ancoră într-un fișier mare. O altfel de abordare este folositoare doar în cazurile în care toată documentația se poate restrânge la un singur fișier HTML împărțit în teme, care este folosit pe toată durata cât aplicația este activă. În acest caz, fișierul este deschis o singură dată la început. Pentru a grupa totuși temele după înțeles, se recomandă crearea unei ierarhii arborescente de directoare și fișiere pe criterii semantice. Un bun exemplu ar putea fi următorul :



14.4.2. Fișierul HelpSet

Atunci când sistemul JavaHelp este activat de către aplicație, primul lucru care este făcut este citirea fișierului HelpSet specificat de aplicație printr-o comandă de activare. Fișierul HelpSet definește pentru aplicație o mulțime de date care descriu sistemul help, unind în fapt toate celelalte metadate, în acest sens fiind fișierul de bază. Fișierul HelpSet include următoarele informații:

Informații	Descriere
fișierul Map	Fișierul Map este folosit pentru a asocia identificatori temelor (care sunt conținute în fișiere HTML) folosind URL-uri (căi spre fișiere)
informații de vizualizare	Informații care descriu navigatoarele care vor fi folosite de HelpSet. Navigatoarele standard sunt: cuprins, index, căutare. Informații despre navigatoarele personalizate sunt incluse, de asemenea, aici.
titlul HelpSet-ului	Numele directorului cel mai de sus din cuprins.
identificatorul Home	Numele identificatorului implicit care este vizualizat, atunci când vizualizatorul de help este chemat fără a specifica un ID în prealabil.
sub-HelpSet-uri	Secțiuni opționale pot fi folosite pentru a include în mod static alte HelpSet-uri, folosind tag-ul <code><subhelpset></code> . HelpSet-urile indicate de acest tag sunt adăugate în mod automat în HelpSet-urile care conțin tag-ul.

După ce aplicația este instalată pe sistemul utilizatorului, pentru a se putea prezenta help-ul, trebuie să fie posibilă găsirea fișierului HelpSet. Aplicația folosește calea spre fișierul HelpSet, pentru a starta sistemul JavaHelp. Prin convenție, numele fișierului HelpSet se termină cu extensia `.hs`. Formatul fișierului HelpSet se bazează pe meta-limbajul de marcare XML (eng. *eXtensible Markup Language*) susținut de Consorțiul Web (W3C) și a cărui specificație o găsiți la adresa : <http://www.w3.org/TR/PR-xml-971208>.

În continuare prezentăm descrierea unui fișier HelpSet - `asistenta.hs` :

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<helpset version="1.0">

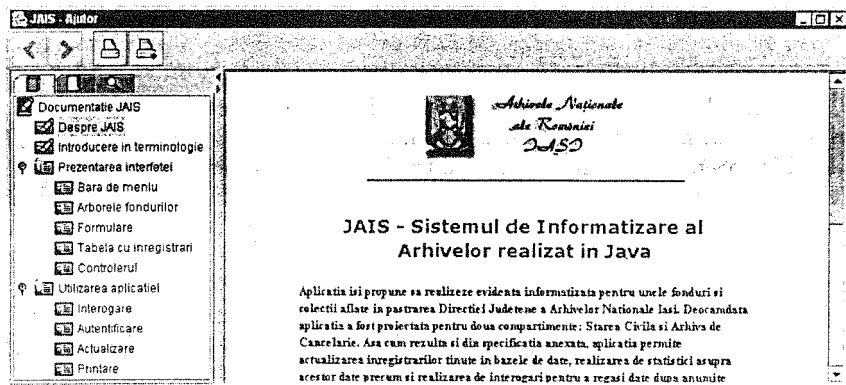
```

```

<!-- titlul ferestrei -->
<title>JAIS - Ajutor</title>
<!-- mapari -->
<maps>
  <homeID>despre</homeID>
  <mapref location="mapare.jhm"/>
</maps>
<!-- vizualizari -->
<view>
  <name>Continut</name>
  <label>Continutul documentatiei</label>
  <type>javax.help.TOCView</type>
  <data>continut.xml</data>
</view>
<view>
  <name>Index</name>
  <label>Indexul documentatiei</label>
  <type>javax.help.IndexView</type>
  <data>index.xml</data>
</view>
<view>
  <name>Cautare</name>
  <label>Cautare in documentatie</label>
  <type>javax.help.SearchView</type>
  <data engine="com.sun.java.help.search.DefaultSearchEngine">
    JavaHelpSearch
  </data>
</view>
</helpset>

```

Acest fișier HelpSet determină vizualizarea următoarei ferestre JavaHelp:



Următoarea tabelă prezintă tag-urile care pot apărea într-un fișier HelpSet:

<helpset>	Tag-ul rădăcină al HelpSet-ului. Poate conține toate celelalte tag-uri.
<title>	Numele ferestrei HelpSet.
<maps>	Specifică tema implicită și URL-ul fișierului Map folosit în HelpSet. Conține următoarele tag-uri: • <homeID> Specifică numele identificatorului implicit a cărui temă asociată va fi afișată în panoul conținut atunci când sistemul help este activat, dacă un identificator de temă nu este specificat explicit. • <mapref> Specifică fișierul Map. De remarcat că în distribuția 1.0 se poate specifica doar un fișier Map. Conține următorul atribut: - location URL-ul fișierului map.
<view>	Definește navigatoarele folosite în sistemul HelpSet. Poate conține următoarele tag-uri: • <name> Numele navigatorului. • <label> Specifică o etichetă asociată cu vizualizarea. Acest șir poate fi accesat de către aplicație și folosit în prezentare. Este afișat la poziționarea mouse-ului pe iconul navigatorului. • <type> Specifică calea spre clasa navigatorului. • <data> Specifică calea spre datele folosite de navigator. Când este folosit de către navigatorul căutare, conține atributul următor: - engine Calea spre motorul de căutare.
<subhelpset>	Acest tag opțional poate fi folosit pentru a specifica HelpSet-urile care vor fi combinate cu HelpSet-ul curent. - location URL-ul fișierului HelpSet care va fi combinat.

14.4.3. Fișierul Map

Așa cum spuneam, când sistemul JavaHelp este activat de către aplicație, primul pas pe care îl va face este să citească fișierul HelpSet. Următorul pas este acela de a citi fișierul Map localizat, folosind fișierul HelpSet. Fișierul map este utilizat pentru a asocia temelor identificatori folosind URL-uri (căi spre fișierele HTML care conțin temele). Prin convenție, fișierele Map folosesc extensia .jhm. Formatul unui fișier

Map se bazează, de asemenea, pe limbajul de marcare XML (W3C). Prezentăm, în continuare, un exemplu de fișier Map - mapare.jhm:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<map version="1.0">
  <mapID target="t.img" url="imagini/t.gif" />
  <mapID target="d.img" url="imagini/d.gif" />
  <mapID target="f.img" url="imagini/f.gif" />
  <mapID target="e.img" url="imagini/e.gif" />
  <mapID target="despre" url="intro/despre.html" />
  <mapID target="introducere.terminologie" url="intro/
    introducere.html" />
  <mapID target="prezentare.introducere" url="prezentare/
    intro.html" />
  <mapID target="prezentare.meniu" url="prezentare/meniu.html" />
  <mapID target="prezentare.arbore" url="prezentare/arbore.html" />
  <mapID target="prezentare.formulare" url="prezentare/formulare.
    html" />
  <mapID target="prezentare.tabela" url="prezentare/tabela.html" />
  <mapID target="prezentare.controler" url="prezentare/controler.
    html" />
  <mapID target="folosire.introducere" url="folosire/introducere.
    html" />
  <mapID target="folosire.interogari" url="folosire/interogari.
    html" />
  <mapID target="folosire.autentificare" url="folosire/autentic.
    html" />
  <mapID target="folosire.actualizare" url="folosire/actualizare.
    html" />
  <mapID target="folosire.printare" url="folosire/printare.html" />
</map>
```

De notat că imaginile folosite (de exemplu, în cuprins) sunt, de asemenea, identificate folosind identificatori. Spre exemplu, fișierului t.gif i-am asociat identificatorul t.img.

În continuare sunt descrise tagurile folosite în fișierele Map:

<map>	Tag-ul rădăcină al fișierului Map. Poate conține tag-uri <mapID> .
<mapID>	Definește o asociere de identificator unei teme. Are următoarele atribute: - target Specifică identificatorul asociat cu URL-ul specificat în atributul url. - url Specifică URL-ul asociat cu identificatorul specificat în atributul target.

14.4.4. Fișierul cuprins

Fișierul cuprins (eng. *Table Of Contents - TOC*) descrie conținutul documentației. Formatul unui fișier cuprins este bazat pe XML și are extensia .xml. Urmează un mic exemplu de fișier cuprins - continut.xml:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE toc PUBLIC
  "- //Sun Microsystems Inc.//DTD JavaHelp TOC Version 1.0
    //EN"
  "http://java.sun.com/products/javahelp/toc_1_0.dtd">
<toc version="1.0">
  <tocitem image="t.jpg" text="Documentatie JAIS"
    target="prima.pagina">
  <tocitem image="e.jpg" text="Despre JAIS" target="despre" />
  <tocitem image="e.jpg" text="Introducere in terminologie"
    target="introducere.terminologie" />
  <tocitem image="d.jpg" text="Prezentarea interfetei grafice"
    target="prezentare.introducere">
  <tocitem image="f.jpg" text="Bara de meniu" target="prezentare.
    meniu" />
  <tocitem image="f.jpg" text="Arborele fondurilor si colectiilor"
    target="prezentare.arbore" />
  <tocitem image="f.jpg" text="Formulare"
    target="prezentare.formulare" />
  <tocitem image="f.jpg" text="Tabela cu inregistrari"
    target="prezentare.tabela" />
  <tocitem image="f.jpg" text="Controlerul"
    target="prezentare.controler"
  />
</tocitem>
  <tocitem image="d.jpg" text="Modul de folosire al aplicatiei"
    target="folosire.introducere">
  <tocitem image="f.jpg" text="Interogare"
    target="folosire.interogari" />
  <tocitem image="f.jpg" text="Autentificare"
    target="folosire.autentificare" />
  <tocitem image="f.jpg" text="Actualizare"
    target="folosire.actualizare" />
  <tocitem image="f.jpg" text="Printare"
    target="folosire.printare" />
</tocitem>
</toc>
```

Acest exemplu determină vizualizarea următorului conținut:

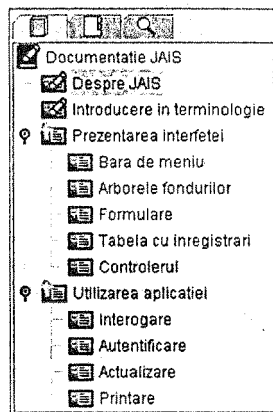


Tabela următoare descrie tag-urile care apar în fișierele TOC:

<toc>	Tag-ul rădăcină al fișierului cuprins. Conține tag-uri <tocitem>.
<tocitem>	Definește o intrare în cuprins. Imbricarea intrarea1 în intrarea2 definește intrarea2 ca fiind inclusă ierarhic în intrarea1. Se pot folosi următoarele atribute: - text Specifică numele intrării. - target (optional) Specifică identificatorul asociat temei care va fi vizualizată, când această intrare va fi aleasă de utilizator. Identificatorii sunt definiți în fișierul Map. - image (optional) Specifică imaginea care va fi afișată în fața intrării din conținut. Argumentul acestui atribut este un identificator definit (asociat cu o imagine GIF sau JPEG) în fișierul Map. Dacă acest atribut nu este specificat, sunt afișate imaginile implicite pentru fișier și director.

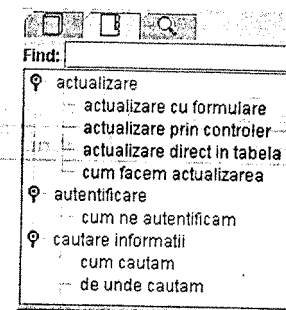
14.4.5. Fișierul index

Fișierul index descrie conținutul navigatorului index și modul în care va arăta acesta. Formatul unui fișier index se bazează pe specificația meta limbajului de marcare XML (W3C). În continuare prezentăm un exemplu simplu de fișier index :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE index PUBLIC
    "-//Sun Microsystems Inc.//DTD JavaHelp Index
    Version 1.0//EN"
```

```
"http://java.sun.com/products/javahelp/index_1_0.dtd">
<index version="1.0">
<indexitem text="actualizare">
<indexitem text="actualizare cu formulare"
    target="prezentare.formulare"/>
<indexitem text="actualizare prin controler"
    target="prezentare.controler"/>
<indexitem text="actualizare direct în tabela"
    target="prezentare.tabela"/>
<indexitem text="cum facem actualizarea"
    target="folosire.actualizare"/>
</indexitem>
<indexitem text="autentificare">
<indexitem text="cum ne autentificam"
    target="folosire.autentificare"/>
</indexitem>
<indexitem text="cautare informatii">
<indexitem text="cum cautam" target="folosire.interogari"/>
<indexitem text="de unde cautam" target="prezentare.formulare"/>
</indexitem>
</index>
```

Acest exemplu produce următoarea vizualizare:



Tag-urile folosite în fișierul index sunt următoarele:

<index>	Tag-ul rădăcină. Poate conține tag-uri <indexitem>.
<indexitem>	Definește o intrare în index. Imbricând intrare1 în intrare2 definim intrare2 ca fiind ierarhic conținută de intrare1. Se folosesc următoarele atribute: - text Specifică textul intrării. - target (optional) Specifică identificatorul temei ce va fi vizualizată, când intrarea este aleasă de utilizator. Identificatorii sunt definiți în fișierul Map.

14.4.6. Indexarea și căutarea

Motorul de căutare al sistemului JavaHelp utilizează o tehnologie de căutare în limbaj natural care regăsește acele pasaje din documentele constituente pentru care răspunsurile la interogări sunt corecte. Tehnologia implică un motor de indexare care analizează documentele pentru a produce un index al conținutului lor și un motor de interogare care utilizează acest index pentru a regăsi pasaje relevante din material. Autorul help-ului este cel care creează baza de date pentru căutare, ce va fi explorată de motorul de căutare al sistemului JavaHelp. Pentru crearea bazei de date se va folosi comanda `jhindexer` din directorul `/javahelp/bin` al pachetului JavaHelp, care are următoarea sintaxă: `jhindexer [opțiuni] [fișier / director]*`. În cazul în care se specifică un director, se vor căuta în interior recursiv toate fișierele conținând teme și se vor indexa, iar dacă se specifică un fișier, el se va indexa.

Prezentăm în continuare lista opțiunilor posibile :

Opțiune	Explicație parametri
-c fișier	Comportarea comenzii poate fi controlată folosind un fișier de configurare; <code>fișier</code> reprezintă numele fișierului de configurare.
-db director	Parametrul <code>director</code> reprezintă directorul în care se va depune baza de date rezultată. Implicit, acesta este numit <code>JavaHelpSearch</code> și este creat în directorul curent.
-locale limba_tara	Constanta <code>limba_tara</code> asociată localizării (limba și țara) determină caracteristica locale a indexului așa cum apare în <code>java.util.Locale</code> . Pentru exemplu: <code>ro RO</code> (româna, România).
-logfile fișier	Capturează în fișierul <code>fișier</code> mesajele generate de <code>hindexer</code> .
-nostopwords	Cuvintele exceptate (eng. <i>stopwords</i>) vor fi indexate și ele.
-verbose	Afișare de mesaje în timpul indexării.

Comanda `c:\j2sdk1.4.0\bin\java -jar c:\jh1.1.3\javahelp\bin\jhindexer.jar` intro prezentare folosire `-c configurare.txt -locale ro_RO -logfile rezultat.txt -verbose` executată în directorul rădăcină al sistemului help determină crearea bazei de date (indexului) formate din 6 fișiere în directorul implicit `JavaHelpSearch`, folosind fișierul de configurare `configurare.txt`, rezultând de asemenea un fișier `rezultat.txt`, în care se scriu informații despre derularea procesului de indexare. Pentru verificarea integrității bazei de date create, se folosește comanda `jhsearch directorul_indexului`, comandă care se află tot în directorul `javahelp/bin`.

Fișierul de configurare oferă următoarele posibilități :

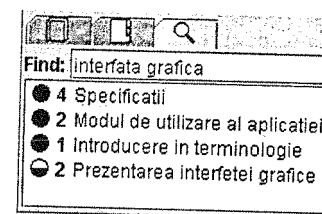
- Memorarea numelor căii spre fișiere cu alt nume în baza de date, opțiune importantă în cazul în care calea spre anumite fișiere din timpul dezvoltării help-ului se schimbă pe parcursul căutărilor. De exemplu: `/proiectare/aplicatia/asistenta/prezentare/meniu.html` va fi memorată sub forma `prezentare/meniu.html`, dacă adăugăm în fișierul de configurare `Index Remove /proiectare/aplicatia/asistenta/; prezentare/meniu.html`

devine `aplicatia/asistenta/prezentare/meniu.html`, dacă adăugăm în fișierul de configurare `IndexPrepend aplicatia/asistenta/`.

Specificarea unei liste de fișiere ce se doresc a fi indexate. De exemplu, adăugând în fișierul de configurare liniile `File meniu.html File tabela.html`, se vor indexa și aceste fișiere.

De asemenea, se poate specifica o listă de cuvinte exceptate. Cuvintele exceptate sunt acele cuvinte care nu se doresc indexate din cauza lipsei lor de importanță în procesul de căutare. Adăugând în fișierul de configurare `StopWords` cuvintele `un, o, toate, toti, sunt, si, oricare`, acestea se vor exclude de la indexare. Același efect l-am fi avut dacă am fi scris în fișierul de configurare : `StopWordsFile` `cuvinte.txt`, unde fișierul `cuvinte.txt` conține aceeași înșiruire de cuvinte de mai sus. Implicit, sunt exceptate o serie de cuvinte din limba engleză (*a, all, am...*). A se vedea ghidul de utilizare.

Pentru a iniția căutarea, utilizatorul introduce o interogare în limbaj natural în navigatorul de căutare în zona de text având eticheta *Find*. Rezultatele sunt raportate înapoi utilizatorului în modul următor:



- Cerculețele din prima coloană indică gradul de potrivire pentru subiectul căutat. Cu cât este mai umplut cercul, cu atât potrivirea este mai bună. Sunt patru grade de potrivire (de la cel mai mare la cel mai mic):



- Numărul din coloana a doua indică de câte ori s-a regăsit potrivirea în tema dată.
- Textul reprezintă numele temei (așa cum este specificat în fișierul conținut).

Motorul de căutare folosește o tehnică numită relaxare graduală (eng. *relaxation ranking*) pentru a identifica și clasifica paginile găsite ca răspunsuri la interogările utilizatorului. Astfel se încearcă găsirea de pasaje în temele help-ului în care, pe cât e posibil, termenii interogării utilizatorului se potrivesc în aceeași formă și în aceeași ordine. Motorul de căutare relaxează automat constrângerile în identificarea paginilor în felul următor:

- nu toți termenii se potrivesc;
- termenii se potrivesc în forme diferite;
- termenii se potrivesc în ordine diferită;
- termenii se potrivesc având cuvinte despărțitoare.

Motorul de căutare asociază penalități corespunzătoare paginilor pentru deviații de la interogarea specificată de utilizator. Procesul de gradare are loc atunci când

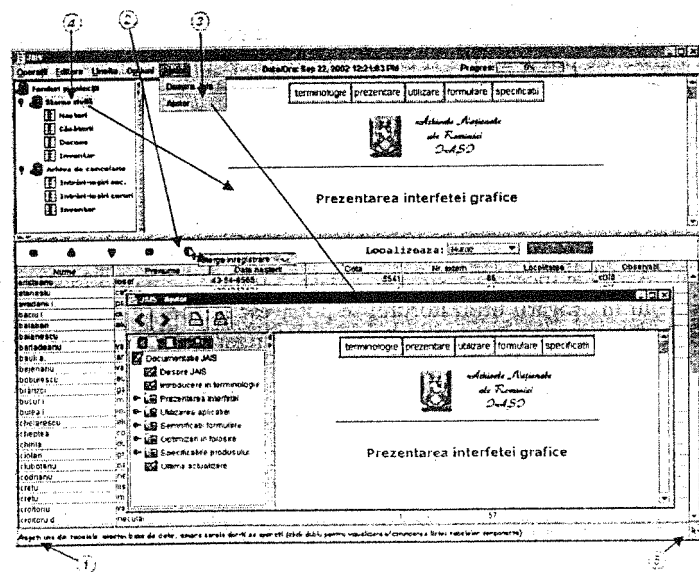
interogările sunt complexe și includ mai mulți termeni. De asemenea, motorul de căutare al sistemului JavaHelp folosește o tehnologie numită morphing pentru a găsi cuvinte cu rădăcini comune.

14.4.7. Compresia și încapsularea

După crearea HelpSet-ului urmează un pas opțional, și anume compresia și încapsularea întregii documentații într-un singur fișier folosind formatul JAR pus la dispoziție de platforma Java. Pentru aceasta trebuie să vă poziționați în directorul rădăcină al sistemului help (în cazul structurii arborescente prezentate mai sus, pe directorul / asistenta) și să apelați comanda `jar -cvf ajutor.jar *` pentru a crea arhiva `dumneavoastră`, care va avea numele `ajutor.jar` și va păstra întreaga structură de directori ale sistemului help creat. Pe parcursul procesului de arhivare veți primi mesaje despre modul în care decurge acest proces. Spre exemplu, pentru fișierul `mapare.jhm` vom fi avertizați: `adding: mapare.jhm (in=5757) (out=2216) (deflated 61%)`, ceea ce înseamnă că fișierul s-a comprimat cu 61%. Pentru a folosi fișierele din arhiva creată nu este nevoie să le extragem, ci le putem utiliza direct folosind protocolul JAR pus la dispoziție de Java 2 SDK.

14.5. Atașarea de help aplicațiilor

Prezentăm, în continuare, diverse modalități de a oferi asistență utilizatorului, ilustrate grafic sugestiv în imaginea care urmează:



1. *Bara de stare* – reprezintă o etichetă (de obicei un obiect `JLabel` căruia i se resetează mereu textul) afișată în partea de jos a ecranului, prin care se pot sugera utilizatorilor informații despre operațiile pe care pot să le facă la un moment dat sau cum s-a desfășurat operația curentă.
2. *Tooltips* – este modalitatea de a adăuga o etichetă explicativă unui buton sau în general, unei componente, care va afișa un mesaj în momentul deplasării mouse-ului pe deasupra ei. Pentru a adăuga tooltip-uri se folosește clasa `JToolTip` sau, mai simplu, metoda `setToolTipText()` asociată fiecărei componente grafice.
3. *Fereastră help* – reprezintă o fereastră nemodală care va fi vizualizată la cererea utilizatorului prin alegerea unei opțiuni dintr-un meniu sau apăsarea unui buton care generează help, respectiv apăsarea tastei `F1`.
4. *Help scufundat în interfața grafică a aplicației* – reprezintă combinarea sau refolosirea componentelor ferestrei sistemului help de la pasul anterior direct în interiorul interfeței grafice a aplicației.
5. *Afișare help la selectarea componentei dorite cu mouse-ul* – prin această metodă trebuie creat mai întâi un buton având următorul aspect: , prin apăsarea căruia se va modifica cursorul, luând forma  cu care mai apoi se selectează componenta al cărei help se vrea vizualizat.

Acest capitol dezbate ultimele trei metode de a adăuga help aplicațiilor, care fac și obiectul sistemului JavaHelp. A se consulta pe parcursul parcurgerii acestui capitol `javadoc`-ul care descrie API-ul JavaHelp. Pentru a adăuga sistemul help aplicației trebuie mai întâi importate clasele care oferă această facilități: `import javax.help.*`. Inițial, în `CLASSEPATH` trebuie adăugată calea spre una din librăriile puse la dispoziție (spre exemplu, `jh.jar`).

Vom discuta mai întâi despre clasele principale pe care sistemul JavaHelp le pune la dispoziție programatorilor și mai apoi modul în care acestea pot fi folosite pentru a rezolva unele aspecte privind adăugarea de help aplicațiilor:

1. Clasa `HelpSet`.
2. Clasa `HelpBroker`.
3. Clasa `CSH`.
4. Help la nivel de fereastră (senzitiv la context).
5. Adăugarea de help componentelor grafice.
6. Butoane sau opțiuni de meniuri care determină vizualizarea help-ului.
7. Help prin selectarea cu mouse-ul.
8. Scufundarea (eng. *embedding*) help-ului în interfața grafică a aplicației.

14.5.1. Clasa `HelpSet`

Legătura dintre aplicație și documentația scrisă sub formă de HTML și structurată folosind XML se realizează printr-o clasă care poartă denumirea `HelpSet`. Această clasă se află în pachetul `javax.help` și este o înfășurătoare pentru conținutul help-ului. Constructorii acestei clase sunt de forma:

- public HelpSet() - creează un HelpSet vid; pentru a-i asocia un HelpSet în mod dinamic se folosește metoda public void add(HelpSet hs); pentru a șterge dinamic un HelpSet se folosește metoda public boolean remove(HelpSet hs).
- public HelpSet(ClassLoader inc, URL helpset) throws HelpSetException - încarcă un HelpSet descris printr-un fișier HelpSet care se află la adresa helpset folosind încărcătorul (eng. loader) de clase inc și returnând eroare dacă nu se poate realiza parsarea fișierului. Dacă inc este null, atunci se va folosi încărcătorul implicit al mașinii virtuale. URL-ul unui HelpSet este returnat de o metodă statică a clasei HelpSet de genul: public static URL findHelpSet(ClassLoader c, String nume), unde c reprezintă încărcătorul de clase, care la fel poate fi null, iar nume reprezintă calea spre fișierul cu extensia .hs.

Exemplu :

```
try {
    URL hsURL=HelpSet.findHelpSet(null,"../asistenta/
    asistenta.hs");
    // hsURL va tine calea spre fisierul HelpSet
    HelpSet hs = new HelpSet(null, hsURL);
    // am creat un obiect HelpSet
} catch (Exception ee) {
    //verificam corectitudinea parsarii
    System.out.println("Eroare la parsare ");
    return;
}
```

14.5.2. Clasa HelpBroker

Obiectul HelpBroker este un agent care negociază și gestionează prezentarea conținutului help pentru aplicație. HelpBroker-ul oferă, de asemenea, metodele necesare pentru a implementa help senzitiv la context. Se poate implementa un sistem help fără a folosi clasa HelpBroker, ceea ce implică scrierea de cod care să-i suplinească lipsa. Se asociază unui obiect HelpSet folosind metoda public HelpBroker createHelpBroker().

```
HelpBroker hb = hs.createHelpBroker();
```

O metodă importantă a acestei clase este public void initPresentation(), care inițializează prezentarea pentru a scurta din timpul de vizualizare. E de preferat utilizarea ei într-un fir de execuție separat la începutul startării aplicației, altfel, în momentul vizualizării se va pierde timpul și cu inițializarea.

Metode pentru gestionarea prezentării	Explicație
public void setDisplay (boolean val_adevar)	Vizualizează fereastra de help și aruncă UnsupportedOperationException în cazul în care operația nu se poate efectua.

Metode pentru gestionarea prezentării	Explicație
public void setViewDisplayed (boolean val_adevar)	Setează dacă apare sau nu panoul de navigare în fereastra JavaHelp.
public void setSize (Dimension dim)	Setează dimensiunea ferestrei JavaHelp; aruncă UnsupportedOperationException dacă operația nu se poate efectua.
public void setLocation (Point punct)	Setează coordonatele la care se va afișa fereastra JavaHelp; la fel aruncă excepția UnsupportedOperationException.
public void setCurrentView (String nume)	Setează navigatorul cu numele cu care este definit în fișierul HelpSet.
public void setCurrentID (String id)	Setează ca topicul care va apărea în panoul de conținut să fie cel ce are identificatorul id; aruncă BadIDException dacă identificatorul nu este definit în fișierul Map.

Metode pentru implementarea help-ului senzitiv la context	Explicație
public void enableHelpKey (Component comp, String id, HelpSet hs)	Setează identificatorul și HelpSet-ul pentru o fereastră JFrame (în acest caz, comp reprezintă root Pane-ul ferestrei) sau pentru o fereastră Window în cazul implementărilor cu AWT (în acest caz comp este chiar fereastra). De asemenea, această metodă determină ca atunci când se apasă tasta Help (tasta F1 implicit) să se realizeze prezentarea help-ului.
public void enableHelp (Component comp, String id, HelpSet hs)	Activează sistemul help pentru componenta comp prin adăugarea identificatorului de temă id și HelpSet-ului hs.
public void enableHelp (MenuItem menu, String id, HelpSet hs)	Același lucru, dar pentru MenuItem-ul menu.
public void enableHelpOnButton (Component comp, String id, HelpSet hs)	Asociază identificatorul de temă și HelpSet-ul pentru o componentă, precum și un ascultător ActionListener, care se execută după activarea componentei și care va vizualiza fereastra help cu topicul dat de identificator (dacă comp nu e AbstractButton sau Button, sau derivată din ele, va arunca excepția IllegalArgumentException).
public void enableHelpOnButton (MenuItem menu, String id, HelpSet hs)	Același lucru, dar pentru un MenuItem.

14.5.3. Clasa CSH

Este o clasă dedicată totalmente implementării help-ului senzitiv la context. Prezентăm următoarele metode statice care permit atașarea de identificatori și de HelpSet-uri componentelor AWT sau Swing, determinând prin apăsarea tastei *Help* să se afișeze help-ul referitor la componenta respectivă:

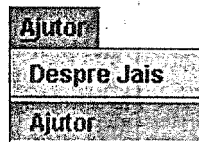
Metode pentru implementarea help-ului senzitiv la context	Explicație
public static void setHelpIDString (Component comp, String helpID)	Asociază componentei grafice comp un identificator definit în fișierul Map. Atunci când se va apăsa F1 și componenta deține focus-ul sau se dă click cu mouse-ul având cursorul  pe componentă, se va afișa fereastra help având drept conținut tema cu identificatorul helpID. Dacă componenta nu are asociată o temă, se va căuta ascendent având drept criteriu relația <i>a part of</i> până se va găsi o asignare.
public static void setHelpSet (Component comp, HelpSet hs)	Asociază HelpSet-ul hs componentei comp.

Aceleași metode sunt valabile pentru MenuItem.

Clasa CSH conține trei clase interioare importante, care sunt de fapt ActionListener-i, ce pot fi asociați componentelor grafice:

- public static class CSH.DisplayHelpFromSource are constructorul public CSH.DisplayHelpFromSource (HelpBroker hb) și preia identificatorul de temă corespunzător componentei grafice pe care o ascultă (în principiu un buton) și vizualizează fereastra help având tema corespunzătoare identificatorului drept conținut. Spre exemplu, secvența următoare determină prin setarea opțiunii *Ajutor* vizualizarea temei asociate acestei opțiuni:

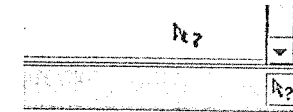
```
JMenuItem opt_ajutor = new JMenuItem("Ajutor");
opt_ajutor.addActionListener(new CSH.DisplayHelpFromSource(hb));
```



- public static class CSH.DisplayHelpAfterTracking are constructorul public CSH.DisplayHelpAfterTracking (HelpBroker hb) și oferă posibilitatea de a obține help prin alegerea cu mouse-ul a componentei dorite. Prin asocierea acestui ascultător (eng. *listener*) unui buton JButton (sau derivat) sau MenuItem, automat prin apăsarea lui se obține cursorul  și butonul rămâne apăsat până la alegerea componentei. După alegere se va afișa fereastra

help corespunzătoare componentei grafice selectate și se va reseta cursorul mouse-ului la starea inițială.

```
context.addActionListener(new CSH.DisplayHelpAfterTracking(hb));
```



- public static class CSH.DisplayHelpFromFocus are constructorul public CSH.DisplayHelpFromFocus (HelpBroker hb) și este folosită pentru a determina vizualizarea help-ului (tema dată de identificatorul asociat, dacă există) pentru componente în momentul în care preiau focus-ul. Este folosit, în general, pentru componente diferite de JFrame sau JWindow.

14.5.4. Adăugare de help ferestrelor

În exemplul următor, ultima linie atașează help ferestrei de bază a aplicației determinând ca apăsarea tastei F1 să afișeze fereastra standard help, a cărei structură a fost descrisă la începutul acestui capitol:

```
HelpBroker hb = hs.createHelpBroker();
//obtinem obiectul HelpBroker din HelpSet-ul hs
//hb.setDisplayed(true); putem vizualiza help-ul oricand
//folosind broker-ul
//in firul de executie urmatore se initializeaza sistemul
//help
Thread fir = new Thread() {
    public void run() {
        try { hb.initPresentation(); }
        catch (Exception e) { System.err.println("Eroare: "+e);}
    }
};
fir.start();
// ....
hb.enableHelpKey(this.getRootPane(), "prima.pagina", null);
//prin apsarea tastei F1 se obtine vizualizarea help-ului
```

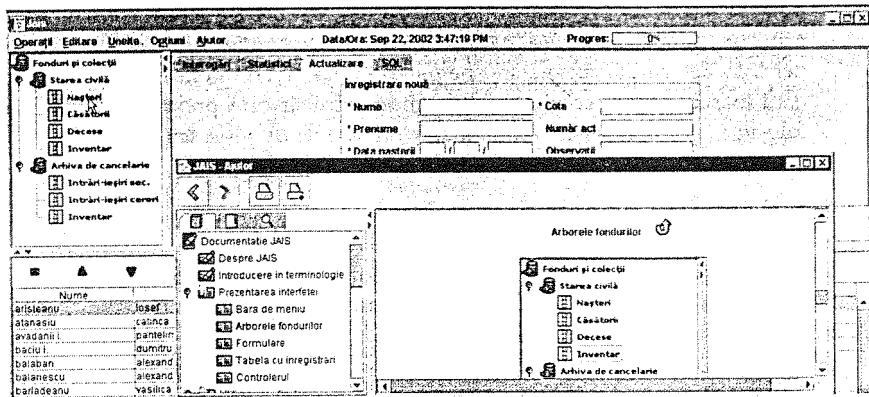
14.5.5. Adăugarea help-ului componentelor

Dacă dorim ca de fiecare dată când apăsăm tasta F1 să obținem informații referitoare la componenta grafică curentă, trebuie să asociem mai întâi componentelor identificatorii descriși în fișierul Map. În exemplul următor adăugăm unui JScrollPane,

care conține un JTree, identificatorul prezentare.arbore descris inițial în fișierul Map:

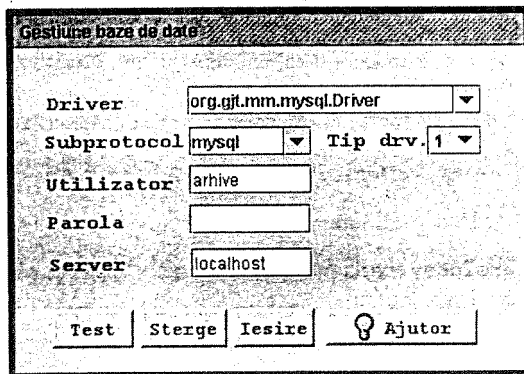
```
derulareArbore= new JScrollPane(arbore);
CSH.setHelpIDString(derulareArbore, "prezentare.arbore");
```

Dacă se apasă tasta F1, atunci când arborele este selectat va apărea fereastra de help având drept conținut tema prezentare.arbore. Chiar dacă arborele nu are asociat identificator, se va afișa tema corespunzătoare JScrollPane-ului datorită căutării ascendente pe baza relației "a part of" de care am pomenit mai sus:



14.5.6. Activarea help-ului prin apăsarea butoanelor

Considerăm următoarea componentă derivată din JDialog :

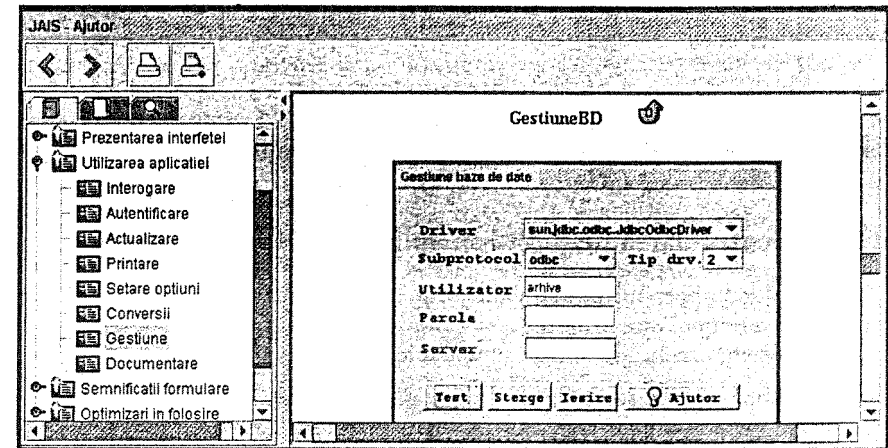


Butonului „Ajutor” i se atașează identificatorul unei teme prin secvența de cod:

```
JButton ajutor = new JButton();
ajutor.setIcon(new ImageIcon(DialogSQL.class.getResource(
    "ajutor.gif")));
```

```
ajutor.setText("Ajutor");
JaisFereastra.hb.enableHelpOnButton(ajutor,
    "folosire.gestiune", null);
```

Atunci când vom apăsa butonul „Ajutor”, se va vizualiza :

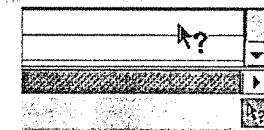


În același mod se procedează și cu JMenuItem.

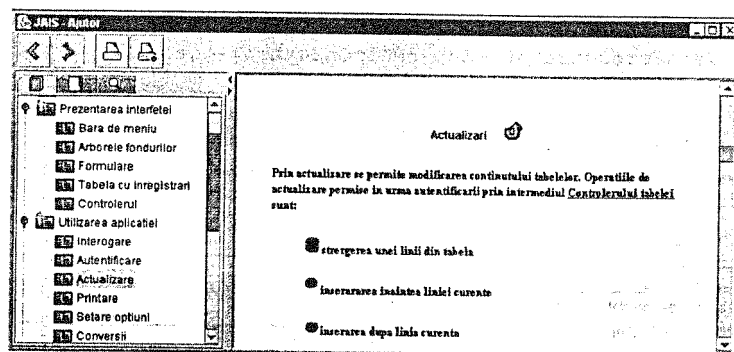
14.5.7. Help activat folosind mouse-ul

Pentru a prezenta help prin poziționarea cu mouse-ul este suficientă secvența de cod următoare:

```
JButton contex=new JButton(new ImageIcon("context.gif"));
//cream un buton
contex.addActionListener(new CSH.DisplayHelpAfterTracking
    (hb));
//asociem ascultator butonului
contex.setToolTipText("Apasati si alegeti componenta dorita");
//adaugam ToolTip butonului
```



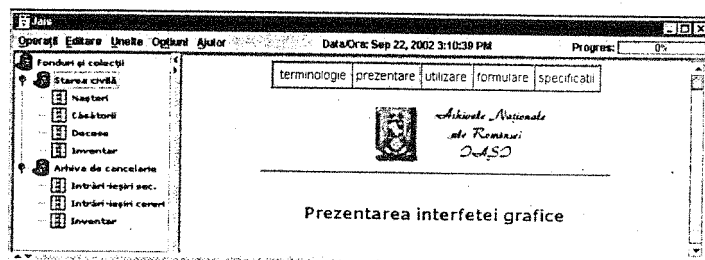
Prin apăsarea butonului mouse-ului, după schimbarea cursorului în poziția curentă, se va vizualiza fereastra de help având următorul conținut (de menționat că tabela are asociat identificatorul folosire.actualizare):



14.5.8. Scufundarea help-ului în aplicație

Putem scufunda porțiuni din sistemul help direct în interfața grafică a aplicației noastre. În exemplul care urmează adăugăm interfeței noastre panoul de conținut al sistemului help, căruia îi setăm și o anumită temă:

```
JHelpContentViewer aj = new JHelpContentViewer(hs);
//aj reprezinta continutul HelpSet-ului hs si este
//un obiect JComponent din faptul ca JHelpContentViewer
//mosteneste clasa JComponent ceea ce ne dreptul
//de a-l adauga obiectelor JContainer sau derivate
JPanel pan=new JPanel(); //cream un panou nou
try {
    aj.setCurrentID("prezentare.introducere");
    // am setat tema dorita
}
catch (Exception en)
{
    System.out.println("am cu ce ba");
}
pan.add(aj); // adaugam continutul intr-un panou
sus.setRightComponent(aj);
// adaugam continutul in partea dreapta a JSplitPane-ului
```



Se pot adăuga aplicației și alte componente ale sistemului help, cum ar fi panoul de navigare. Evităm astfel apariția pe ecran a unei alte ferestre dedicate help-ului, care ne-ar putea stânjeni.

14.6. Concluzii

Pentru început s-au prezentat noțiuni introductive legate de sistemul de asistență pentru aplicații, urmate de o descriere a pachetului JavaHelp. Ca observație, acest pachet (JavaHelp) nu a fost introdus încă în nici o distribuție a platformei Java, din pricina modificărilor continue prin care trece. Este prezentată în continuare alcătuirea standard a interfeței sistemului JavaHelp și a principiilor pe care acest sistem le respectă.

Într-o manieră progresivă, sunt prezentați pașii necesari pentru a crea sistemul de asistență al unei aplicații. S-a folosit drept fundal un exemplu efectiv, ușurând astfel înțelegerea conceptelor implicate în acest proces. Pentru fiecare document descriptor XML folosit s-a prezentat librăria de taguri.

Ultima secțiune a capitolului descrie metodele efective care pot fi folosite pentru a adăuga Help aplicațiilor. Sunt enumerate clasele folosite în acest proces și exemple efective de folosire a lor.

14.7. Test grilă

Întrebarea 14.7.1. Care este funcționalitatea sistemului JavaHelp?

- Oferă posibilitatea creării Java-Doc-ului pentru aplicațiile Java.
- Oferă posibilitatea de a adăuga ajutor contextual aplicațiilor.
- Reprezintă documentația oficială a limbajului Java.

Întrebarea 14.7.2. Ce pachete trebuie incluse în CLASSPATH pentru a putea crea sistemul Help pentru aplicațiile Java?

- Numai jsearch.jar.
- Pachetele jhall.jar, jhbasic.jar și jh.jar.
- Pachetul jh.jar este îndeajuns.

Întrebarea 14.7.3. Ce reprezintă scufundarea ferestrelor Help în aplicație?

- Ascunderea ferestrei de bază Help după fereastra de bază a aplicației.
- Integrarea Help-ului direct în aplicație.
- Renunțarea la sistemul Help pentru aplicație.

Întrebarea 14.7.4. Care este succesiunea corectă de pași necesari în realizarea sistemului pentru o aplicație?

- Atașarea de help aplicației, crearea temelor HTML, scrierea descriptorilor XML.
- Crearea temelor HTML, scrierea descriptorilor XML, atașarea de Help aplicației.

- c) Scrierea descriptorilor XML, crearea temelor HTML, atașarea de Help aplicației.

Întrebarea 14.7.5. Care dintre următoarele afirmații sunt adevărate?

- Fișierul Map este folosit pentru a asocia identificatori temelor.
- Fișierul cuprins (Table Of Contents – TOC) descrie conținutul documentației.
- Fișierul index conține indexul temelor HTML.

Întrebarea 14.7.6. Care este rolul utilitarului jhindexer?

- Permite crearea fișierului index.
- Permite regăsirea de intrări în fișierul index.
- Afișează rezultatele unei căutări în fișierul index.

Întrebarea 14.7.7. Fie secvența de cod:

```
JButton c=new JButton(new ImageIcon("context.gif"));
c.addActionListener(new CSH.DisplayHelpAfterTracking(hb));
context.setToolTipText("Apasati si alegeti componenta dorita");
```

Care dintre afirmațiile următoare este adevărată?

- Apăsarea butonului determină terminarea aplicației.
- Apăsarea butonului determină schimbarea formei cursorului.
- După apăsarea butonului, la prima apăsare a butonului stâng al mouse-ului se va afișa Help contextual.

14.8. Exerciții propuse spre implementare

Exercițiul 14.8.1. Prezentați acest capitol sub forma unui sistem JavaHelp.

Exercițiul 14.8.2. Adăugați ajutor contextual aplicațiilor dumneavoastră, urmând pașii descriși pe parcursul acestui capitol.

Exercițiul 14.8.3. Personalizați după preferință interfața standard a sistemului JavaHelp folosind informațiile din ghidul de utilizare.

14.9. Proiecte propuse spre implementare

Proiectul 14.9.1. Realizați o aplicație având interfață grafică și care să permită crearea într-un mod facil a sistemului Help pentru o aplicație Java. Pentru implementare se va folosi librăria Swing.

Proiectul 14.9.2. Creați o aplicație care să permită asocierea de Help aplicațiilor într-o manieră comodă. Se va permite încărcarea și parcurgerea surselor pentru adăugarea de Help contextual.

Proiectul 14.9.3. Creați o aplicație care să permită prezentarea documentației Java-doc într-o manieră asemănătoare JavaHelp-ului.

Proiectul 14.9.4. Realizați un applet care să ofere help contextual aplicațiilor prin intermediul rețelei.

15. Internaționalizarea aplicațiilor

15.1. Cuvinte cheie

- internaționalizare
- localizare
- seturi de caractere
- formătări

15.2. Introducere

Internaționalizarea este procesul de proiectare a aplicațiilor, astfel încât acestea se pot adapta la diverse limbi și regiuni fără a fi nevoie de modificări în aplicație după ce ciclul de dezvoltare al acesteia s-a terminat. Un bun exemplu în acest sens este cazul unui applet care detectează localizarea mașinii pe care se execută și prezintă informațiile celor care îl accesează în limba lor. Un alt exemplu ar putea fi aplicațiile care au meniuri și, în general, interfața realizată în mai multe limbi, utilizatorul selectând limba dorită fie la instalarea aplicației, fie pe parcursul utilizării ei. La fel ne putem gândi la sistemul help atașat unei aplicații care este de preferat să fie în mai multe limbi, înțelegerea lui fiind esențială utilizatorilor aplicației. Deseori termenul „internaționalizare” este abreviat prin *i18n*, deoarece sunt 18 caractere între prima literă „i” și ultima literă „n”.

În acest context, apare noțiunea de „localizare” care reprezintă tocmai procesul de structurare a datelor pe diverse limbi și regiuni, ceea ce reprezintă un pas important în adaptarea unui program pentru diverse locații. Prin localizare (în sens de locație) vom înțelege o regiune geografică sau politică în care se vorbește aceeași limbă. Procesul de localizare a unei aplicații presupune traducerea textelor etichetelor, mesajelor de eroare și a help-ului, respectiv formatarea datelor corespunzătoare respectivelor locații, precum ar fi valorile monetare, datele calendaristice, numerele. Termenul „localizare” mai este abreviat prin *l10n*, deoarece sunt 10 litere între „l” și „n”.

Mai vechiul JDK 1.0 nu oferă suport pentru internaționalizare. JDK 1.0 poate mânui numai primele 256 caractere din codarea Unicode, care corespund caracterelor din limba engleză. Nu există posibilitatea transferului (conversiei) spre alte codificări diferite de Unicode. Internaționalizarea a fost pentru prima oară adăugată în 1.1 și de aceea se poate considera că platforma 1.1 este limita de la care se poate vorbi despre internaționalizarea aplicațiilor. De asemenea, din experiență se poate observa că este de preferat ca o aplicație să fie construită de la început respectând principiile internaționalizării, deoarece procesul de transformare a acesteia pentru a fi în acord cu internaționalizarea este complicat și înseamnă rescriere de cod.

Un program internaționalizat are următoarele caracteristici:

- elementele text, cum ar fi mesajele din bara de stare sau conținutul etichetelor, nu sunt încapsulate în aplicație, ci sunt ținute în afara ei, de unde sunt încărcate în mod dinamic;
- suportul pentru noi limbi nu implică recompilarea;
- datele „culturale” precum data, sistemul de măsuri apar în conformitate cu regiunea în care se află utilizatorul și cu limba pe care acesta o vorbește.

Din lista de cerințe prezentată mai sus se poate observa o încercare de a separa datele sensibile (cele care se doresc prezentate conform unei anumite localizări) de codul aplicației, pentru a evita recompilările acesteia. De asemenea, se observă necesitatea unei metode prin care să se realizeze descrierea diverselor localizări. Pachetul `java.util` oferă suportul pentru aceste două aspecte prin intermediul clasei

`ResourceBundle`, care stă la baza separării datelor de aplicație, respectiv clasa `Locale`, care oferă un mod standard de stabilire a localizărilor. Descrierea modului în care trebuie folosite clasele `Locale` și `ResourceBundle` este prezentată în secțiunile acestui capitol. Mai întâi vom discuta însă despre modul în care limbajul Java folosește seturile de caractere.

15.3. Codări

O mulțime de codări de caractere (eng. *coded character set*) reprezintă o asociere (eng. *mapping*) între o mulțime abstractă de caractere și o mulțime de numere întregi. *US-ASCII*, *ISO 8859-1*, *Unicode (ISO 10646-1)* sunt exemple de mulțimi de codări de caractere.

O schemă de codare a caracterelor (eng. *character-encoding scheme*) reprezintă o mapare între o mulțime de codări de caractere și o mulțime de secvențe de octeți. *UTF-8* (Universal Transformation Format folosit de protoalele de rețea și de sistemul de fișiere Unix), *UCS-2*, *UTF-16*, *EUC* sunt exemple de astfel de scheme. În general, o schemă este asociată unei singure mulțimi de codări, purtând numele acesteia, așa cum este cazul *US-ASCII*, care reprezintă și numele setului de caractere și al schemei de codare. Un exemplu în care schema și mulțimea de caractere nu au același nume ar putea fi cazul schemei *UTF-8*, care este folosită pentru a coda *Unicode*. Alte scheme sunt asociate cu mai multe mulțimi de caractere. Astfel, schema *EUC* poate fi folosită pentru a coda diverse mulțimi de caractere din zona asiatică. Schemele de codare în notația *Microsoft* poartă numele de „code page”, numele lor începând cu prefixul „cp”, spre exemplu *Cp1250* pentru Europa de Est.

Vom folosi în general noțiunea de „set de caractere” (eng. *character set*) atât pentru mulțimea de codări, cât și pentru schemă, cititorul desprinzând din context despre care dintre acestea este vorba. În Java, numele seturilor de caractere trebuie să înceapă cu o literă sau o cifră și nu contează dacă sunt scrise cu litere mari sau mici. Numele seturilor de caractere respectă convențiile specificate prin *RFC 2276*. Setul de caractere nativ pentru limbajul Java este *Unicode* versiunea 3.0, această alegere rezolvând multe dintre problemele legate de manipularea și reprezentarea caracterelor. Intern, caracterele sunt reprezentate folosindu-se schema de codare *UTF-16*. De asemenea, limbajul Java suportă conversii între *Unicode* și alte seturi de caractere. Aceste conversii se realizează prin intermediul claselor `java.io.InputStreamReader`, `java.io.OutputStreamWriter` și `java.lang.String` și sunt necesare, spre exemplu, pentru interacțiunea aplicațiilor Java cu bazele de date, alte aplicații etc. care folosesc alte seturi de caractere. În cazul interacțiunii cu astfel de aplicații, se vor face conversiile datelor spre *Unicode*, vor fi prelucrate și mai apoi transformate din nou spre setul de caractere original.

15.3.1. Despre Unicode

În limbajul de programare Java variabilele, având tipul `char`, reprezintă caractere Unicode. Unicode reprezintă un standard de codare a caracterelor pe 16 biți (doi octeți) care oferă suport pentru cele mai multe dintre limbile existente, permițând codarea a 2^{16} caractere. Mai vechiul standard ASCII (eng. *American Standard Code for Information Interchange*) folosește codarea pe 7 biți, iar *ISO-Latin-1* folosește codarea pe 8 biți. Chiar dacă sursele programelor sunt scrise de cele mai multe ori folosind aceste ultime două codări, înainte de a se trece la procesarea lor în JVM, acestea sunt translatate în Unicode. Specificația Unicode o puteți găsi la adresa www.unicode.org, ea reprezentând documentul de referință pentru cei ce folosesc acest mod de codare a caracterelor.

Nu vom insista prea mult asupra setului de caractere Unicode, pentru că, în general, nu este nevoie de aceasta pentru a lucra cu limbajul Java, codarea fiind folosită intern, iar trecerile spre alte codări făcându-se prin clase specializate. Vom prezenta, în continuare, numai câteva aspecte ale acestei codări.

Unicode, privit ca set de caractere, este alcătuit din 17 planuri (eng. *planes*), fiecare putând conține 65.536 caractere. În total se pot ține 1.1120.64 caractere. Primul plan (notat Plane 00) este unul special, fiind numit BMP (eng. *Basic Multilingual Plane*), și conține majoritatea caracterelor și suportul pentru o mare diversitate de limbi. Celelalte planuri sunt numite planuri suplimentare (eng. *supplementary planes*) și sunt folosite efectiv abia începând cu versiunea Unicode 3.1 (ultima versiune existentă la momentul scrierii acestei cărți). În tabela următoare prezentăm structura Unicode 3.1:

Plan	Nume plan	Caractere
0 (0x00)	Basic Multilingual Plane (BMP)	49,196
1 (0x01)	Supplementary Multilingual Plane pentru scripturi și simboluri (SMP)	1,594
2 (0x02)	Supplementary Ideographic Plane (SIP)	43,253
14 (0x0E)	Supplementary Special-purpose Plane (SPP)	97

Unicode Versiunea 3.1 este descrisă prin două standarde ISO: *ISO 10646-1:2000* (Partea 1: Arhitectura și Basic Multilingual Plane) și *ISO 10646-2:2000* (Partea 2: Supplementary Planes).

Ultima versiune Unicode permite trei tipuri de codări: *UTF-8*, *UTF-16*, *UTF-32*. Numărul de biți din denumiri reprezintă numărul de biți din care este alcătuită unitatea cu care se face codarea. Spre exemplu, pentru *UTF-8* caracterele se pot coda variabil prin unu, doi, trei sau patru octeți. În cazul *UTF-16*, caracterele se pot coda prin unul sau două cuvinte (eng. *word* = 2 octeți). În cazul *UTF-32*, codarea se face fix prin secvențe de 32 biți.

15.3.2. Seturi de caractere

Compilerul Java și alte unelte Java pot procesa numai fișiere conținând codarea Latin-1, eventual combinată cu notația Unicode (secvențe `/u` pentru caractere. Pentru a transforma fișierele dintr-o codare oarecare în Latin-1 se poate folosi o unealtă numită `native2ascii`, care se află în directorul `/bin` al distribuției Java. Sintaxa acestei comenzi este:

```
native2ascii [optiuni] [fișier intrare [fișier iesire]]
```

Dacă fișierul de ieșire este omis, se va trimite către ieșirea standard (monitor). Dacă fișierul de intrare lipsește, se va citi de la intrarea standard (tastatură). Opțiunile pot fi:

- `reverse` realizează operațiunea inversă transformând fișiere scrise folosind setul Latin-1 cu secvențe în notație Unicode spre o altă codare;
- `encoding nume_codare` specifică numele codării folosită în procesul de conversie, iar numele este dat de tabela cu codări acceptate pe care o găsiți în `java-doc: docs\guide\intl\encoding.doc.html`. Dacă nu se specifică nici o codare, se va lua în calcul codarea implicită.

Spre exemplu, putem să salvăm ca text simplu (eng. *plain*), cu numele `unicode.txt`, un document redactat în Word folosind diacriticele (ultimile versiuni de Word permit selectarea mai multor codări în momentul salvării). Presupunem că îl salvăm în Unicode (UTF8). Pentru a-l converti la Latin-1 vom folosi comanda:

```
native2ascii -encoding UTF-8 unicode.txt latin.txt
```

În Java, orice set de caractere are asociat un nume canonic care poate fi diferit de numele din specificații și o mulțime de aliasuri, iar gestionarea lor se face prin intermediul clasei `java.nio.charset.Charset`. Pachetul `java.nio` (denumirea vine de la New I/O) reprezintă noua librărie pentru lucrul cu intrări-ieșiri pentru o aplicație Java. Acest pachet mai aduce o clasă nouă, și anume `java.nio.charset.spi.CharsetProvider`, care poate fi folosită pentru a adăuga noi seturi de caractere folosind extensii. Prezentăm, în continuare, seturile de caractere de bază care sunt suportate prin convenție de orice platformă Java:

Nume canonic	Descriere
US-ASCII	Șapte biți ASCII (ISO646-US), numit și blocul de bază Latin al setului de caractere Unicode.
Cp 1252	Windows Latin-1.
ISO-8859-1	ISO Latin Alphabet No. 1 (ISO-LATIN-1)
ISO-8859-15	Alfabetul Latin. Nr. 9.
UTF-8	UCS Transformation Format pe opt biți.
UTF-16BE	UCS Transformation Format pe șaisprezece biți, ordinea big-endian.
UTF-16LE	UCS Transformation Format pe șaisprezece biți, ordinea little-endian.
UTF-16	UCS Transformation Format pe șaisprezece biți.

J2SDK 1.4 și J2RE (pentru Windows, doar versiunea internațională) suportă și multe alte seturi de caractere (în general, specifice anumitor regiuni), în mod diferit pentru pachetul `java.io` și `java.lang`. Prin „suportă” înțelegem că permite realizarea de conversii pentru că intern, după cum spuneam, întotdeauna se lucrează cu Unicode. Pentru informații complete puteți consulta documentul aflat în `java-doc-ul limbajului Java`, în directorul `j2sdk1.4.0\docs\guide\intl\encoding.doc.html`.

Pentru a determina seturile de caractere suportate de mașina virtuală se poate folosi metoda statică `availableCharsets()` din clasa `java.nio.charset.Charset`:

```
import java.util.*;
import java.nio.charset.*;
//...
SortedMap sm=Charset.availableCharsets();
Set c=sm.keySet();
Object[] ob=c.toArray();
for (int i = 0; i < ob.length; i++)
    System.out.print(ob[i].toString()+" ");
```

Se va obține acest rezultat: ISO-8859-1 ISO-8859-15 US-ASCII UTF-16 UTF-16BE UTF-16LE UTF-8 windows-1252.

Întotdeauna există o codare implicită care este selectată automat de mediul de lucru Java, în funcție de sistemul de operare peste care este instalată platforma și de localizarea setată pentru acesta. Spre exemplu, în cazul în care avem setată localizarea Romanian pentru Windows, implicit se va considera setul de caractere *Cp1250*, care reprezintă setul de caractere pentru estul Europei. Codarea implicită este importantă deoarece face legătura între mediul de execuție Java, în care caracterele sunt reprezentate prin Unicode, și sistemul de operare, care folosește pentru sistemul de fișiere o altă codare (care va deveni codarea implicită). Codarea implicită este păstrată de proprietatea `file.encoding` din `System` și se obține printr-un apel de forma:

```
System.out.println("Codarea implicita: "+System.getProperty
("file.encoding"));
```

Suportul pentru transformări între diverse codări se realizează prin intermediul claselor `java.io.PrintStream`, `java.io.InputStreamReader`, respectiv `java.io.OutputStreamWriter`.

`PrintStream` reprezintă fluxul pentru ieșire care oferă diferite suprascrieri convenabile ale metodei `print()`, folosit în mod uzual pentru scrierea către ieșirea standard. Acesta are un constructor care permite folosirea altor seturi de caractere diferite de setul implicit de caractere. Prezentăm modul în care se poate seta ca spre ieșirea standard să se scrie cu caractere est-europene (dacă localizarea este Romanian, deja această codare este folosită implicit):

```
PrintStream ps1=new PrintStream(System.out , true, "Cp1250");
System.setOut(ps1);
```

În exemplul care urmează prezentăm redirectarea ieșirii standard spre un fișier având numele `iesire.txt`. De asemenea, scrierea se va face folosind setul de caractere pentru Europa de Est.

```
PrintStream ps=new PrintStream(new FileOutputStream (new File
("iesire.txt") , true, "Cp1250");
System.setOut(ps);
```

Pentru a vizualiza rezultatul scris în fișier trebuie folosit un editor care vizualizează caracterele est-europene (poate folosi codarea *Cp1250*). Altfel (pentru că editoarele simple folosesc în general ASCII), „ș” și „ț” nu se vor vizualiza corect.

Clasa `InputStreamReader` reprezintă o punte între fluxuri de biți și fluxuri de caractere. Poate fi folosită și pentru a transforma textul non-Unicode considerat a aparține unui set de caractere și reprezentat prin fluxul de biți în text Unicode. Dacă nu este specificat explicit prin convenție, se consideră setul de caractere implicit. Un exemplu de transformare a unui text ASCII în text Unicode:

```
FileInputStream fisier = new FileInputStream("intrare.txt");
InputStreamReader uni = new InputStreamReader(fisier,
"US-ASCII");
String codare = uni.getEncoding();
Metoda getEncoding() specifică codarea folosită de flux.
```

Pentru trecerea din Unicode în alte seturi de caractere se poate folosi clasa `OutputStreamWriter`. Exemplul următor permite scrierea unui șir de caractere într-un fișier folosind codarea *UTF8*:

```
try {
    FileOutputStream fisier = new FileOutputStream("text.txt");
    Writer iesire = new OutputStreamWriter(fisier, "UTF8");
    iesire.write("salut");
    iesire.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

15.4. Setarea localizării

Un program internaționalizat poate vizualiza informații diferite în funcție de locația din care este apelat. Spre exemplu, dacă este vorba de un applet, acesta va prezenta informații în limba română, dacă este rulat pe o mașină din România, și în limba franceză, dacă este apelat pe o mașină din Franța. În limbajul Java, localizarea clienților aplicațiilor se încapsulează într-un obiect `Locale` din pachetul `java.util`. Obiectele `Locale` sunt simpli identificatori pentru diverse combinații de limbi și regiuni, neconținând atribute caracteristice localizării. Unei aplicații Java nu i se

atribuie un singur obiect `Locale`, ci clasele se pot conforma unei instanțe `Locale`, în acest caz numindu-se *local-senzitive* (eng. *local-sensitive*). Din acest motiv putem avea într-o aplicație clase conform diferitelor instanțe `Locale` și deci informațiile pot fi prezentate în mai multe limbi concomitent. Dacă nu se specifică o instanță `Locale` pentru obiectele *local-senzitive*, atunci implicit aceasta este stabilită de JVM (eng. *Java Virtual Machine*) în momentul startării ca fiind obiectul `Locale` corespunzător platformei gazdă. O referință spre acesta puteți obține printr-un apel `Locale.getDefault()`. De asemenea, se poate seta obiectul `Locale` implicit printr-un apel `Locale.setDefault(Locale l)`, dar trebuie să avem grijă, pentru că el va fi adoptat implicit de toate instanțele claselor *local-senzitive* ale aplicației.

Modul de creare a obiectelor `Locale` este extrem de simplu. Spre exemplu, pentru a crea un obiect `Locale` corespunzând limbii franceze și țării Canada vom folosi un constructor de forma `Locale loc = new Locale("fr", "CA")`, iar pentru limba română și România, un apel `Locale loc = new Locale("ro", "RO")`. Primul șir reprezintă o pereche de litere mici corespunzând limbii conform standardului *ISO-639* a cărui specificație o găsiți la adresa:

<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>.

Cel de al doilea șir format dintr-o pereche de litere majuscule reprezintă codul țării și respectă standardul *ISO-3166* a cărui specificație o puteți găsi la adresa :

http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html.

Clasa `Locale` mai pune la dispoziție un constructor prin care se permite introducerea unui al treilea parametru care poartă numele de cod variabil. De obicei acesta este folosit pentru a specifica referințe la platforma pe care rulează aplicația. Spre exemplu, putem folosi un apel `Locale l = new Locale("ro", "RO", "UNIX")`. Acest parametru este opțional și dacă apare obiectul care îl folosește va ști cum să-l trateze. Un obiect `Locale` poate fi creat doar pentru o anumită limbă, în acest caz folosind la construcția lui un apel în care al doilea parametru este un șir null ca, de exemplu, `new Locale("ro", null)`. Clasa `Locale` pune la dispoziția utilizatorilor o serie de constante corespunzând anumitor limbi și țări, cum ar fi `Locale.FRANCE`, `Locale.ITALY` etc.

Metoda	Descriere
<code>Public String getCountry()</code>	Codul ISO al țării.
<code>Public String getLanguage()</code>	Codul ISO al limbii.
<code>public final String getDisplayLanguage()</code> <code>public String getDisplayLanguage (Locale loc)</code>	Returnează un șir reprezentând numele limbii corespunzătoare localizării curente. Dacă se precizează ca parametru o instanță <code>Locale</code> , șirul returnat va fi numele limbii localizării curente în localizarea loc.
<code>public final String getDisplayCountry()</code> <code>public String getDisplayCountry (Locale inLocale)</code>	Returnează un șir reprezentând numele țării corespunzătoare localizării curente. Dacă se precizează ca parametru o instanță <code>Locale</code> , șirul returnat va fi numele țării localizării curente în localizarea loc.

O dată creat un obiect `Locale`, se pot face interogări asupra lui pentru a obține informații folosind comenzile prezentate în tabelul anterior.

Pentru a ilustra modul de lucru cu localizările, prezentăm următoarea secvență de cod:

```
Locale loc=Locale.getDefault();
System.out.println("Localizare implicita "+loc.toString());
System.out.println(loc.getLanguage());
System.out.println(loc.getCountry());
System.out.println(loc.getDisplayLanguage());
System.out.println(loc.getDisplayCountry());
System.out.println(loc.getDisplayLanguage(Locale.FRENCH));
System.out.println(loc.getDisplayCountry(Locale.FRENCH));
```

Dacă localizarea implicită este `ro_RO`, atunci va fi afișat următorul mesaj:

```
Localizare implicita: ro_RO
ro
RO
română
România
romain
Roumanie
```

Localizarea implicită poate fi schimbată fie programatic, fie prin setarea unei localizări pentru sistemul de operare. Spre exemplu, pentru Windows acest lucru se face din *Start/Settings/Control Panel/Regional Settings*. Dacă vom seta, spre exemplu, aici localizarea *English (United States)*, în urma execuției secvenței de mai sus va fi afișat următorul conținut:

```
Localizare implicita: en_US
en
US
English
United States
anglais
Etats-Unis
```

Pentru a vedea care sunt localizările suportate de clasele *local-senzitive*, acestea pun la dispoziția programatorilor metoda `getAvailableLocales()`, care returnează o listă a localizărilor. Spre exemplu, printr-un apel `NumberFormat.getAvailableLocales()` vom obține o listă a localizărilor pe care această clasă le suportă. De asemenea, clasa `Locale` oferă posibilitatea determinării tuturor localizărilor instalate:

```
import java.util.*;
import java.text.*;
```



```

public class LocalizariAcceptate {
    static public void main(String[] argumente) {
        Locale lista[] = Locale.getAvailableLocales();
        for (int i = 0; i < lista.length; i++) {
            System.out.println(lista[i].toString());
        }
    }
}

```

Va rezulta o înșiruire de reprezentări ale localizărilor de forma: ar_EG, be_BY, bg_BG, ca_ES, ..., ro, ro_RO, ... etc. Toate localizările sunt suportate în același fel și independent de sistemul de operare peste care este instalată mașina Java și de localizările acestuia. Singura implicare a sistemului de operare este în selectarea localizării implicite și a datei și orei implicite, care vor fi cele ale sistemului de operare. J2SDK 1.4 suportă un mare număr de localizări (inclusiv, ro și ro_RO), așa cum se poate vedea și în pagina documentației `java-doc\docs\guide\intl\locale.doc.html`. Mediul de execuție J2RE este distribuit în două versiuni, una suportând același set de localizări ca și J2SDK, iar alta numai localizările English/United States.

15.5. Separarea datelor

Datele sensibile la localizare vor fi păstrate separat de aplicație, grupate (eng. *bundle*) în funcție de diferitele localizări. Spre exemplu, textul vizualizat de interfața grafică intră în categoria datelor sensibile la localizare. Un buton va afișa eticheta OK pentru limba engleză, *De acord* pentru limba română și o altă etichetă pentru o altă limbă. Această afișare diferențiată nu se face printr-o tratare în codul aplicației, ci prin simpla alegere a instanței `Locale` pentru obiectul `JButton`. Gruparea datelor pe diferite localizări se realizează prin intermediul clasei `ResourceBundle`.

Conceptual, `ResourceBundle` reprezintă un set de subclase având drept nume o aceeași rădăcină urmată de numele unei localizări în formatul obișnuit (`codLimba_codTara_codVariant`). Fiecare astfel de subclasă ține toate datele specifice unei localizări particulare grupate sub formă de proprietăți.

Exemple de astfel de subclase:

```

Eticheta
Eticheta_ro_RO
Eticheta_en_US
Eticheta_en_GB

```

Pentru a obține o referință spre o astfel de subclasă, corespunzătoare unei anumite localizări, se poate folosi secvența care urmează:

```

Locale localizare = new Locale("ro", "RO");
ResourceBundle resursa = ResourceBundle.getBundle("Eticheta",
    localizare);

```

În interior, fiecare subclasă `ResourceBundle` este structurată ca un vector de perechi cheie/valoare împreună cu metodele de acces (eng. *accessors*) spre acestea. O cheie identifică unic un element din colecția păstrată într-un obiect `ResourceBundle` și este întotdeauna un șir de caractere. O valoare este în general un șir, dar poate fi orice fel de obiect. Pentru o cheie se poate obține valoarea corespunzătoare printr-un apel de forma `getString("cheie")` sau `getObject("cheie")`. Pentru cea de a doua metodă trebuie să avem grijă să realizăm operația de cast spre clasa care dă tipul obiectului.

Platforma Java 2 oferă două subclase `ListResourceBundle` și `PropertyResourceBundle`, care extind clasa abstractă `ResourceBundle`, fiind implementări efective ale acesteia, ușor de folosit de programatori. Dacă cele două clase nu îndeplinesc cerințele dezvoltatorilor (spre exemplu, timpi prea leneși de execuție), aceștia pot să creeze propria clasă pentru gestionarea datelor sensibile la localizare prin extinderea clasei `ResourceBundle`. Pentru aceasta trebuie suprascrise două metode ale acesteia, și anume `handleGetObject(String cheie)`, care returnează valoarea corespunzătoare cheii `cheie`, și `getKeys()`, care va returna o listă `Enumeration` a cheilor resursei. Prezentăm, în continuare, o astfel de implementare simplă care păstrează două chei, pentru mai multe chei fiind nevoie de o structură (cel mai bine s-ar preta o tabelă `HashMap`):

```

// limba romana
public class ResursaProprie extends ResourceBundle {
    public Object handleGetObject(String cheie) {
        if (cheie.equals("da")) return "De acord";
        if (cheie.equals("nu")) return "Anuleaza";
        return null;
    }
}

// limba engleza
public class ResursaProprie_en extends ResursaProprie {
    public Object handleGetObject(String cheie) {
        if (cheie.equals("tastaDeAcord")) return "Ok";
        return null;
    }
}

```

Atunci când se dorește crearea unui buton, se va folosi un apel de forma:

```

JButton buton = new JButton(myResources.getString("da"));

```

Dacă am fi folosit forma clasică, nu am mai fi oferit suport aplicației pentru internaționalizare:

```

JButton buton = new JButton("De acord");

```

Dacă dorim să nu realizăm propriile noastre implementări ale clasei `ResourceBundle`, putem folosi o extensie a acesteia, și anume clasa `PropertiesResourceBundle`,

care are în spate un fișier text având numele clasei și extensia `.properties` și conținând perechi cheie/valoare unde și cheia, și valoarea sunt șiruri de caractere. Dacă se dorește păstrarea de valori care sunt instanțe ale clasei `Object`, este indicată folosirea clasei `ListResource`. Fișierul text poate fi creat cu orice editor de texte, dar trebuie să aibă numele clasei `ResourceBuddle` și să aibă extensia `properties`. Această clasă respectă principiile clasei `Properties`, care reprezintă un set de proprietăți persistente (salvate pe suport). Întotdeauna trebuie creat un fișier implicit. Spre exemplu, pentru un buton prezentăm un exemplu de fișier implicit:

```
# fișierul ResourceBundle Buton.properties
da= DeAcord
nu= Anuleaza
```

Comentariile se scriu folosind simbolul `"#"`. Proprietățile se scriu una sub alta în forma cheie=valoare.

Pe lângă fișierul implicit se vor crea noi fișiere cu proprietăți pentru alte localizări, conținând valorile traduse pentru aceleași chei. Sursa aplicației nu se va modifica, deoarece accesul spre valoarea dependentă de localizare se va face doar prin intermediul cheii. Numele fișierului este format din numele fișierului implicit la care se adaugă localizarea și extensia `.properties`. Spre exemplu, resursele pentru limba engleză vor fi ținute într-un fișier `Button_en.properties`:

```
# fișierul ResourceBundle Button_en.properties
da= OK
nu= CANCEL
```

Următoarea secvență descrie procesul de creare a unor butoane folosind fișiere cu proprietăți:

```
Locale localizare= new Locale("ro","");
ResourceBundle text = ResourceBundle.getBundle("Button_ro",
localizare);
String mesajDa=text.getString(da);
JButton butonDa= new JButton(mesajDa);
String mesajNu=text.getString(nu);
JButton butonNu= new JButton(mesajNu);
```

Printr-un apel al metodei `getBundle(String numeResursa, Locale localizare)` se va căuta mai întâi dacă există definită o clasă `ResourceBuddle` având numele `numeResursa`, iar dacă nu se găsește, se va căuta un fișier de proprietăți care are numele corespunzător. Pentru căutarea fișierelor se folosește un algoritm descris în `java-doc`, ultimul luat în calcul fiind fișierul implicit. Dacă nu se găsește nici un fișier de proprietăți, atunci se va arunca o excepție `MissingResourceException`.

Pentru compatibilitate cu modul în care a fost implementat sistemul de proprietăți (clasa `Properties`), fișierele cu proprietăți corespunzătoare clasei `PropertyResourcesBuddle` trebuie scrise folosind setul de caractere *Latin-1*, eventual cu secvențe *Unicode*.

Vom prezenta, în continuare, clasa `ListResourceBuddle`, reprezentând o altă metodă de a gestiona resursele, folosind de această dată fișiere `.class` salvate pe suport. Pentru fiecare localizare se va crea câte o clasă extinzând `ListResourceBuddle` și având un nume corespunzător, care va fi mai apoi compilată, rezultând un fișier cu extensia `class`. Clasele vor trebui să suprascrie metoda `getContent()` care returnează un tablou ale cărui elemente sunt perechi cheie-valoare.

```
public class Butoane_ro extends ListResourceBuddle {
    public Object[][] getContents() {
        return continut;
    }
    static final Object[][] continut =
    {{ "da", "De acord"}, {"nu", "Anuleaza"}, {"cod", new Integer
    (40)} };
}

public class Butoane_fr extends ListResourceBuddle {
    public Object[][] getContents() {
        return continut;
    }
    static final Object[][] continut =
    { {"nu", "Annuler"}, {"stop", "Arreter"}, {"cod", new
    Integer(20)} };
}
```

După compilare vor rezulta două fișiere `Butoane_ro.class` și `Butoane_fr.class`. Dacă sursele `java` ale acestor clase sunt scrise cu un alt set de caractere, într-un editor de texte care permite acest lucru, acesta trebuie specificat explicit în momentul compilării, pentru a se face corect conversia spre `Unicode`. Altfel, se consideră că sursele au fost scrise folosind setul de caractere al sistemului de operare (spre exemplu, pentru `Windows`, pentru localizarea *English*, setul *Latin-1*). Pentru a compila sursa scrisă folosind diacritice românești, se va folosi:

```
javac -encoding Cp1250 Butoane_ro.java
```

Un lucru demn de remarcat este faptul că pentru toate clasele, proprietățile se moștenesc astfel încât nu e nevoie să repetăm proprietățile dacă sunt aceleași pentru mai multe localizări. Ierarhia este dată de numele pe care îl poartă grupurile de resurse, această ierarhie determinând algoritmul de căutare ascendentă a proprietăților.

15.6. Formatări

Diversele localizări implică și folosirea diferitelor formătări pentru datele ce vor fi expuse utilizatorilor. Numerele, sistemul monetar, datele calendaristice, reprezentarea orei și mesajele sunt cele mai importante tipuri de date care suportă procesul de formatare. La baza formătării în limbajul Java stă clasa `Format` din pachetul `java.text`, din care sunt derivate toate celelalte clase folosite pentru formatare.

15.6.1. Formatarea numerelor

Numerele sunt reprezentate diferit în lume. Spre exemplu, în Statele Unite, numărul 3,140 reprezintă trei mii o sută patruzeci, iar, în România, acest număr reprezintă trei și o sută patruzeci. Echivalentul în România pentru acest număr este 3.140. Aplicațiile internaționalizate vor oferi utilizatorilor formatarea corectă a numerelor în funcție de localizarea curentă. Diferențele constau în simbolul zecimal, separatorii pentru grupuri, numărul de cifre după simbolul zecimal și modul de folosire a cifrei zero înainte de simbolul zecimal.

Clasa folosită pentru formatarea numerelor este `java.text.NumberFormat`. Pentru a obține o instanță a acestei clase corespunzătoare unei localizări se va folosi un apel `NumberFormat.getNumberInstance(Locale localizare)`. Având instanța `NumberFormat`, formatarea se poate realiza printr-un apel al metodelor `format(long număr)` sau `format(double număr)` pentru a obține un șir de caractere conținând numărul formatat, șir care va fi prezentat utilizatorilor.

Secvența care urmează, aplicată pentru mai multe localizări, va determina afișarea diferitelor formătări pentru un același număr:

```
Locale localizare; // o localizare oarecare
Double numar = new Double(67947.123);
NumberFormat formatare = NumberFormat.getNumberInstance(
    localizare);
String rezultat = formatare.format(numar);
System.out.println(rezultat + " " + localizare.toString());
67 947,123    fr_FR
67.947,123    de_DE
67,947.123    en_US
67.947,123    ro_RO
```

15.6.2. Reprezentarea monetară

Pentru fiecare localizare, sumele de bani se vor reprezenta într-un anumit mod. De asemenea, simbolurile monetare sunt diferite de la o țară la alta. Formatarea sumelor de bani se realizează în același mod ca și formatarea numerelor, cu singura deosebire

că se va folosi un apel `getCurrencyInstance()`. De remarcat faptul că se face doar afișarea unei aceleași sume în diferite localizări, fără a face conversiile valutare corespunzătoare.

```
Locale localizare; // o localizare oarecare
Double numar = new Double(67947.123);
NumberFormat formatare = NumberFormat.getCurrencyInstance(
    localizare);
String rezultat = formatare.format(numar);
System.out.println(rezultat + " " + localizare.toString());
67 947,123 F    fr_FR
67.947,123 DM   de_DE
$67,947.123     en_US
67.947,123 LEI  ro_RO
```

15.6.3. Formatarea datei calendaristice și a orei

Datele calendaristice sunt păstrate intern, independent de orice localizare, sub formă de obiecte `Date`, dar afișarea acestora se poate face doar ca șiruri de caractere conform unei anumite localizări. Pentru a obține anul, luna, săptămâna dintr-un obiect `Date` se va folosi un obiect `Calendar`. Aceste operații, chiar dacă inițial au fost incluse, au devenit depreciate pentru clasa `Date`, deoarece această clasă nu a fost proiectată pentru a oferi suport pentru internaționalizare. Formatarea datei încapsulate într-un obiect `Date` se realizează cu ajutorul clasei `java.text.DateFormat` într-o manieră asemănătoare cu formatarea numerelor.

```
Locale localizare; // o localizare oarecare
Date azi = new Date();
DateFormat formatare = DateFormat.getDateInstance(DateFormat.
    DEFAULT, localizare);
String data = formatare.format(azi);
System.out.println(data + " " + localizare.toString());
```

Efectul acestei secțiuni de cod în cazul mai multor localizări îl puteți vedea în continuare:

```
07.01.2003    ro_RO
7 janv. 2003 fr_FR
Jan 7, 2003   en_US
```

Afișarea datei se poate face cu mai multe stiluri. Clasa `DateFormat` suportă mai multe stiluri, și anume `DEFAULT`, `SHORT`, `MEDIUM`, `LONG`, `FULL` introduse ca parametru de intrare pentru metoda `getDateInstance()`.

Pentru formatarea orei se va folosi metoda `getTimeInstance()`, ca în exemplul care urmează:

```
DateFormat
formatare=DateFormat.getTimeInstance(DateFormat.DEFAULT, localizare);
```

Clasele `java.text.SimpleDateFormat` și `java.text.DateFormatSymbols` pot fi folosite pentru a crea formatare proprii pentru date calendaristice, formatare care să fie folosite, spre exemplu, în localizările proprii.

15.6.4. Formatarea mesajelor

Această secțiune se ocupă de formatarea mesajelor compuse. Prin mesaj compus vom înțelege un mesaj care conține atât părți fixe, cât și părți variabile. Vom marca părțile variabile ale unui mesaj prin subliniere:

Peste 300.000 de persoane au vizitat situl Web www.infoiasi.ro până la data de 1 ianuarie 2002.

Părțile variabile ale unui mesaj compus pot fi date, șiruri, numere, unități monetare și procente. Pentru a formata un mesaj compus independent de localizare, se va construi un șablon care va fi matrița cu care se creează un obiect `MessageFormat`. Șablonul are un anumit mod de construcție, conform descrierii pe care o găsiți în clasa `MessageFormat`.

Prezentăm un exemplu de șablon pentru mesajul compus prezentat anterior, pe care presupunem că îl păstrăm în clasa `Mesaje`, extensie a clasei `ResourceBundle`:

```
sablon= Peste {2, number, integer} persoane au vizitat
situl Web {0} până la data de {1, date, long}
```

O instanță a clasei `Mesaje` se obține astfel:

```
ResourceBundle mesaje = ResourceBundle.getBundle("Mesaje",
new Locale("ro", "RO"));
```

Se observă că acest mesaj este o reprezentare a șirului inițial, în care s-au înlocuit secvențele variabile cu acolade, având drept conținut numărul secvenței variabile urmat de tipul ei, respectiv modul de formatare. Se poate observa că nu este o anumită ordine a numerotării secvențelor variabile.

Pentru a crea un obiect `MessageFormat` se va folosi:

```
MessageFormat formatare = new MessageFormat("");
formatare.setLocale(new Locale("ro", "RO"));
```

Pentru a configura obiectul formatare astfel încât să se poată formata folosind șablonul depozitat în clasa `Mesaje`, acesta din urmă se va extrage din `ResourceBundle` printr-un apel `getString()` și, mai apoi, va fi folosit printr-un apel `applyPattern()`, așa cum se poate vedea în secvența care urmează:

```
formatare.applyPattern(mesaje.getString("sablon"));
```

Introducerea datelor variabile în șir se va realiza prin încapsularea lor într-un vector de obiecte, ca în exemplul următor:

```
Object[] argumenteVariabile={"www.infoiasi.ro", new Date(),
new Integer(300000)};
```

Ordinea în care se introduc elementele în tablou este importantă și trebuie să fie în concordanță cu șablonul depozitat în `ResourceBundle`. Pentru a popula un șablon cu date variabile, se vor folosi apeluri de forma:

```
String rezultat = formatare.format(argumenteVariabile);
```

Va rezulta exact șirul inițial. Foarte interesant este faptul că, stocând un șablon într-o clasă `ResourceBundle`, îi putem aplica foarte ușor aceleași date inițiale. Putem, spre exemplu, să afișăm foarte ușor un mesaj în altă limbă decât cea implicită, cu formatarea corespunzătoare. Spre exemplu, folosind șablonul `Over {2, number, integer} people have visited {0} website since {1, date, long}` obținem șirul `Over 300000 people have visited www.infoiasi.ro website since January 14, 2003`, folosind secvența de cod:

```
Locale localizare1=new Locale("en", "US");
ResourceBundle mesaj1 =ResourceBundle.getBundle("Mesaje",
localizare1);
formatare1.setLocale(localizare1);
formatare1.applyPattern(mesaj1.getString("sablon"));
String rezultat1 = formatare1.format(argumenteVariabile);
System.out.println(rezultat1);
```

15.7. Lucrul cu fonturi

În ceea ce privește fonturile, platforma Java dă posibilitatea utilizării a două tipuri de fonturi, și anume fonturi fizice și fonturi logice. Fonturile fizice pot fi *True Type* sau *PostScript Type 1*. JRE caută fonturile fizice fie în directorul cu fonturi al sistemului de operare (spre exemplu, `Windows/Fonts` pentru sistemul de operare `Windows`), fie în directorul `/lib/fonts` din directorul în care este instalat JRE. În unul dintre aceste două directoare utilizatorii pot să-și instaleze propriile fonturi.

Implicit, platforma Java 2 oferă o serie de fonturi fizice din familia *Lucida Sans*. Acestea vin împreună cu JRE și nu sunt dependente de sistemul de operare. Folosirea lor oferă posibilitatea unui look-and-feel identic pentru toate sistemele de operare peste care este instalată platforma Java. Acestea sunt „*Lucida Sans*”, „*Lucida Sans Typewriter*” și „*Lucida Bright*” și permit folosirea stilurilor regulat, bold, italic, italic-bold, fiind instalate în directorul `lib/fonts`.

Fonturile logice reprezintă mapări peste fonturile fizice deja existente. În general, un font logic mapează mai multe fonturi fizice, oferind astfel un set mai mare de caractere la un moment dat. Mapările sunt păstrate în fișiere având numele `font.properties` stocate în directorul `/lib` al JRE. Din această categorie de fonturi, orice implementare a platformei Java trebuie să ofere suport pentru *Serif*, *SansSerif*, *Monospaced*, *Dialog* și *DialogInput*. Utilizatorul poate interveni asupra unor fișiere `font.properties` deja existente sau poate să creeze altele, după propriile preferințe.

Utilizatorul poate afla, la un moment dat, toate fonturile disponibile pentru mediul de lucru curent și poate utiliza fonturile folosind numele lor, ca, de exemplu,

Times Roman sau *Taboma*. De asemenea, se pot folosi fonturi neinstalate încă, folosind apeluri ale metodei `create()` a clasei `Font`. Pentru mai multe informații puteți consulta secțiunea destinată desenării din capitolul „Swing”.

15.8. Introducerea textului

Keyboard Layout și *Input Method* sunt cele două modalități de a introduce text drept conținut pentru o componentă text într-un mod mai complex decât simpla afișare a caracterului corespunzător tastei apăsată. Acest lucru este valabil pentru cazul în care un caracter nu poate fi reprezentat printr-o singură apăsare de tastă. Astfel, se pot introduce toate caracterele unei limbi, care pot fi un număr considerabil, folosind tastatura care are un număr finit de taste.

În cazul *Keyboard Layout* se realizează o remapare a tastelor pentru a se putea introduce alte caractere, diferite de cele desenate pe taste. În Java, sistemul pentru *Keyboard Layout* este total dependent de sistemul de operare peste care este instalată platforma. Spre exemplu, în Windows, schimbarea *Keyboard Layout* se va realiza din fereastra *Settings/Control Panel/Keyboard/Language* sau din *Language Bar*, dacă aceasta este disponibilă (eventual folosind shortcut-uri definite în prealabil). Dacă de aici se va alege tastatura română, atunci orice componentă Swing va accepta introducerea diacriticelor prin apăsarea tastelor corespunzătoare. Trebuie reținut faptul că componentele Swing codează textul folosind Unicode, astfel încât programatorul este responsabil, mai apoi, pentru modul în care transferă textul înspre și dinspre componentele Swing, pentru a nu apărea probleme de codare.

În ceea ce privește *Input Method*, se poate spune că există mai multe metode de a introduce text de la tastatură. Prin metoda directă (eng. *on the spot, inline*) se permite introducerea unei secvențe de caractere direct în componentele text, dar marcate prin subliniere, astfel că scrierea lor efectivă va avea loc numai după o confirmare. Acesta este modul implicit de introducere pentru componentele care au asociată o metodă de introducere. Prin metoda indirectă (eng. *below the spot*), se va activa o fereastră auxiliară pentru compunerea textului (eng. *composition*), în imediata apropiere a poziției curente a cursorului (neieșirea ei din ecran este asigurată). Prin apăsarea unei combinații de taste prestabilite, caracterele introduse în fereastra de compoziție sunt convertite într-o altă secvență sau un singur caracter care va fi transferat în componenta text.

Java 2 oferă posibilitatea atât a folosirii metodelor de introducere oferite de sistemul de operare, cât și a definirii de către dezvoltatori a propriilor metode de introducere a textului. Utilizatorilor le este pus la dispoziție un întreg cadru (eng. *framework*) de lucru în acest ultim scop, care poartă denumirea de *Input Method Framework*. Acest cadru cuprinde un API client conținut în pachetul `java.awt.im` prin care se poate realiza programatic comunicarea dintre componente și ferestrele de compoziție corespunzătoare metodei curente de introducere a datelor. De asemenea, mai cuprinde un motor SPI cuprins în pachetul `java.awt.im.spi` care oferă posibilitatea creării de metode de introducere independente de o anumită aplicație Java, care vor fi, mai apoi, folosite în diferite medii de rulare prin simpla atașare a fișierului `jar`

care le conține în directorul `/lib/ext`. La baza acestui cadru stă o specificație pe care o puteți găsi în ghidul atașat documentației standard (`/docs/guide/imf`) a JRE.

Metodele de introducere pot fi folosite nu numai pentru introducerea de caractere pentru diverse limbi, ci și în alte scopuri. A se vedea modalitatea prin care utilizatorul este asistat în procesul de scriere a codului într-un mediu vizual. În momentul editării, atunci când numele metodei a fost scris, prin apăsarea unei combinații de taste va apărea o listă derulantă a tuturor supraîncărcărilor cunoscute ale metodei, dintre care utilizatorul poate alege una.

Exemple de metode de introducere puteți găsi în directorul documentației: `/docs/guide/imf/spi_sample/index.html`.

15.9. Pașii necesari pentru internaționalizarea aplicațiilor

Pentru orice aplicație există anumite caracteristici care au probabilitatea cea mai mare de a fi subiectul internaționalizării, și anume:

- Mesajele
- Etichetele componentelor grafice
- Ajutorul on-line
- Sunetele
- Culorile
- Graficele
- Iconurile
- Datele calendaristice
- Timpul
- Numerele
- Unitățile monetare
- Unitățile de măsură
- Numerele de telefon
- Titlurile personale și onorifice
- Adresa poștală
- Poziționarea în pagină

Aceste informații sunt izolate, în general, de aplicație (nu sunt codate în aceasta) tocmai pentru a o putea internaționaliza.

În momentul internaționalizării, apar și anumite probleme. Spre exemplu, deoarece setul de caractere poate cuprinde și diacritice, nu se mai pot face comparații pentru stabilirea literelor de genul:

```
if ((caracter >= 'a' && caracter <= 'z')) { }
```

Determinarea literelor, cifrelor, semnului pentru spațiu se realizează folosind standardul Unicode prin metode statice de genul `isCharacter(ch)`, `isDigit(ch)`, `isSpaceBar(ch)` oferite de clasa `Character`.

De asemenea, compararea șirurilor nu se mai poate realiza prin metodele clasei `String equals()` și `compareTo()` atunci când dorim să realizăm sortări, din cauza

caracterelor noi care apar în alfabetele diferite de cel englez. În aplicațiile internaționalizate, comparațiile se realizează prin intermediul clasei `Collator`, mai precis prin intermediul metodei `compare()`. Spre exemplu, pentru compararea șirurilor conținând caractere românești:

```
Locale diacritice=new Locale("ro", "RO");
Collator pentruRomana = Collator.getInstance(diacritice);
Int rezultat = pentruRomana.compare("păr", "par");
```

Pentru internaționalizarea aplicațiilor care nu au fost gândite în acest sens de la început, aveți la dispoziție o unealtă standard pe care o puteți descărca gratuit de pe situl Sun www.javasoft.com.

15.10. Concluzii

Pentru început au fost descrise aspecte generale legate de internaționalizarea (i18n) și localizarea aplicațiilor (l10n). S-a discutat mai apoi despre seturi de caractere, realizarea de conversii între acestea, despre codări și, în special, despre Unicode, care reprezintă setul de caractere folosit intern de mașina virtuală Java.

S-au pus mai apoi problemele localizării și a modului în care acest principiu este codat în aplicațiile Java prin intermediul clasei `Locale`. Mai apoi s-a vorbit despre modalitatea de a despărți datele de codul aplicației, pentru a putea astfel adapta aplicația la mai multe localizări, prin folosirea clasei `ResourceBundle` și mai ales a derivatelor din aceasta pe care platforma Java le pune la dispoziția dezvoltatorilor.

Am vorbit despre formatarea datelor conform localizărilor, și anume formatarea numerelor, unităților monetare, datelor calendaristice și a orei, respectiv formatarea mesajelor.

Am descris câteva aspecte legate de fonturi și metodele de introducere a textului, fără a intra în amănunte, rămânându-le cititorilor să studieze API-urile corespunzătoare.

15.11. Test grilă

Întrebarea 15.11.1. Ce reprezintă internaționalizarea aplicațiilor?

- Scrierea de aplicații care să permită folosirea meniurilor scrise în limba română.
- Un proces de proiectare a aplicațiilor astfel încât acestea să poată accesa fișiere scrise cu diverse seturi de caractere.
- Un proces de proiectare a aplicațiilor astfel încât acestea să se poată adapta la diverse localizări.

Întrebarea 15.11.2. Ce reprezintă Unicode?

- Un set de caractere.
- Un criptosistem.
- O aplicație folosită pentru internaționalizarea altor aplicații.
- Cod binar.

Întrebarea 15.11.3. Care este funcționalitatea utilitarului `native2ascii`?

- Convertește fișiere scrise folosind codarea `UTF-16` în fișiere folosind codarea `UTF-8`.
- Conversie spre setul *Latin-1* din orice alt set de caractere.
- Localizează fișierele având numele drept parametru.
- Traduce în *Unicode* sursele încărcate în mașina virtuală.

Întrebarea 15.11.4. Fie secvența de cod:

```
Locale.setDefault(new Locale("ro", "RO"));
Locale loc=new Locale("en", "US");
System.out.println("Localizare"+loc.toString());
```

Ce se va afișa în urma execuției codului?

- Localizare: ro_RO.
- Localizare: româna:Romania.
- Localizare: en_US.
- Se va arunca o excepție.

Întrebarea 15.11.5. Presupunând doar un fișier având numele `Siruri.properties` conținând următoarea informație:

```
da= DeAcord
nu= Anuleaza
```

și secvența de cod:

```
Locale localizare= new Locale("ro","Ro");
ResourceBundle text = ResourceBundle.getBundle("Buton",
localizare);
System.out.println("Mesaj:"+text.getString(da)+" / "+ text.
getString(nu));
```

Ce se va întâmpla după execuția codului?

- Se va afișa o excepție `MissingResourceException`.
- Se va afișa „Mesaj: DeAcord/Anuleaza”.
- Se va afișa „Mesaj: OK/Cancel”.

Întrebarea 15.11.6. Fie secvența de cod:

```
Locale localizare=Locale.getDefault();
Float numar=new Float(845.123);
NumberFormat f=NumberFormat.getCurrencyInstance(localizare);
System.out.println(f.format(numar));
```

În urma execuției codului, care dintre efectele prezentate în continuare este valabil întotdeauna?

- Se va afișa 845,123.
- Se va afișa: 845,123 LEI.

- c) Se va arunca o excepție.
- d) Se va afișa suma 845.123, în localizarea implicită.
- e) Se va afișa numărul real 845.123, în localizarea implicită.

Întrebarea 15.11.7. Ce reprezintă metodele de introducere a textului?

- a) Metode prin care putem introduce diacritice.
- b) Metode prin care putem mapa alte caractere tastaturii.
- c) Metode de a introduce date în aplicațiile Java.
- d) Metode prin care putem formata textul din componentele Swing.

15.12. Exerciții propuse spre implementare

Exercițiul 15.12.1. Realizați o aplicație Java care să facă conversii între diversele codări (*UTF-8*, *UTF-16*, *UTF-32*) ale setului de caractere Unicode.

Exercițiul 15.12.2. Să se scrie o clasă Java care suprascrie *ResourceBundle*, permițând salvarea grupurilor de date în fișiere XML, de unde vor putea mai apoi fi folosite.

Exercițiul 15.12.3. Scrieți o aplicație care face conversii ale fișierelor scrise cu orice tip de caractere spre fișiere scrise folosind Unicode.

15.13. Proiecte propuse spre implementare

Proiectul 15.13.1. (Dicționar) Realizați o aplicație sau applet având funcționalitatea unui dicționar român-englez, englez-român. Acesta va ține datele separat de cod, sub formă de fișiere XML sau în baze de date. Va oferi posibilitatea introducerii de noi cuvinte, modificarea vechilor cuvinte etc.

Proiectul 15.13.2 (Translator). Creați o aplicație sau applet care să traducă expresii standard în mai multe limbi. Se vor folosi formatarea mesajelor și seturi de caractere specifice fiecărei limbi în parte.

Proiectul 15.13.3. Creați o aplicație (sau un applet) care să poarte conversații inteligibile cu cei ce o accesează. Aplicația va detecta singură limba în care i se adresează clientul și va fi destul de instruită pentru a învăța cuvinte și expresii introduse de acesta.

Proiectul 15.13.4. Realizați o aplicație care să reprezinte un plug-in pentru orice altă aplicație, permițând introducerea de diacritice românești folosind secvențe Unicode. Pentru documentație, veți consulta *Input Method Framework*.

Proiectul 15.13.5. Realizați o aplicație care să reprezinte un gestionar de resurse. Va permite gruparea acestora și crearea lor, iar mai apoi includerea lor în aplicații într-un mod uniform.

Proiectul 15.13.6. Realizați o aplicație care să permită internaționalizarea surselor Java care nu au fost proiectate de la început conform principiilor internaționalizării. Aplicația va determina șirurile de caractere din surse și va oferi posibilitatea salvării acestora în fișiere, împreună cu traduceri ale lor pentru diverse localizări, de unde vor fi mai apoi încărcate folosind *ResourceBundle*, în funcție de localizarea folosită curent de aplicație.

Bibliografie

Referințe cărți:

- [Ben90] – M. Ben-Ari: *Principles of Concurrent and Distributed Programming*, Prentice Hall (1990).
- [BeP99] – Bell D., Perr M.: *Java for Students*, second edition, Prentice Hall (1999).
- [Bud00] – Budd, T.: *Understanding Object-Oriented Programming with JAVA*, Addison-Wesley (2000).
- [Bur01] – Buraga, S.: *Tehnologii Web*, Matrix Rom (2001).
- [Bur02] – Buraga, S.: *Proiectarea siturilor Web*, Polirom (2002).
- [BTOT] – Buraga, S., Tarhon-Onu, V., Tanasă, Ș.: *Programare Web în bash și Perl*, Polirom (2002).
- [CGI00] – Chan, M.C., Griffith, S.W., Iasi, A.F.: *JAVA. 1001 secrete pentru programatori*, Teora (2000).
- [Eck98] – Eckel, B.: *Thinking in JAVA*, Prentice Hall (1998).
- [GJS96] – Gosling, J., Joy, B., Stelle, G.: *The Java Language Specification*, USA (1996).
- [GrK97] – Grand, M., Knudsen, J.: *JAVA. Fundamental Classes Reference*, O'Reilly (1997).
- [Hor00] – Horton, I.: *Beginning Java 2. JDK1.3*, third edition, Wrox Press Ltd (2000).
- [HSH99] – Hughes, M., Shoffner, M., Hamner, D.: *JAVA. Network Programming*, Manning (1999).
- [LaO99] – Lambert, K.A., Osborne, M.: *Java. A Framework for Programming and Problem Solving*, PWS Publishing (1999).
- [NoS96] – Norton, P., Stanek, W.: *Peter Norton's Guide to Java Programming*, Samsa Press (1996).
- [RHE99] – Roberts, S., Heller, P., Ernest, M.: *JAVA 2 Certification Study Guide*, Sybex (1999).
- [Vad99] – Văduva, C.M.: *Programarea în JAVA*, Microinformatica (1999).
- [Van96] – Vandenburg, G.L. et al.: *Tricks of the Java Programming Gurus*, Samsa Publishing (1996).
- [WiN96] – Winston, P.H., Narasimhan, S.: *On to JAVA*, Addison-Wesley (1996).

Referințe Web:

- Buraga, S., *Proiectarea interfețelor utilizator*: <http://www.infoiasi.ro/~busaco/teach/courses/interfaces/index.html>
- Eckel, B., *Thinking in Java*, third edition: <http://www.mindview.net/Books>
- Wilson, S., Violet, S., Haase, C., *High Performance GUIs with the JFC/Swing API*: <http://developer.java.sun.com/developer/community/chat/JavaLive/2003/j10121.html>
- * * *, *Web Services Tutorial*: <http://java.sun.com/webservices/docs/1.0/tutorial/index.html>

- ***, *Java 2 Platform, SE v1.4.1 API documentation*:
<http://java.sun.com/j2se/1.4.1/docs/api/index.html>
- ***, *Servlet Documentation*:
<http://java.sun.com/products/servlet/docs.html>
- ***, *JSP Documentation*: <http://java.sun.com/products/jsp/docs.html>
- ***, *The Source for Java(TM) Technology*: <http://java.sun.com>
- ***, *Java World*: <http://www.javaworld.com>
- ***, *Javalobby*: <http://www.javalobby.org>
- ***, *Application Development Trends*: <http://www.javareport.com>
- ***, *W3C Technical Reports and Publications*: <http://www.w3c.org/TR>
- ***, *Extensible Markup Language (XML)*: <http://www.w3c.org/XML/>
- ***, *The Extensible Stylesheet Language (XSL)*: <http://www.w3c.org/Style/XSL/>
- ***, *JDBC Tutorial*:
<http://java.sun.com/docs/books/tutorial/jdbc/index.html>
- ***, *JDBC API Documentation*:
<http://java.sun.com/j2se/1.4.1/docs/guide/jdbc/index.html>
- ***, *Swing, tutorial*:
<http://java.sun.com/docs/books/tutorial/uiswing/index.html>
- Robinson, M., Vorobiev, P., *Swing Release Java 2*:
<http://developer.java.sun.com/developer/Books/swing2/>
- ***, *Tutorial de grafică 2D*:
<http://java.sun.com/docs/books/tutorial/2d/index.html>
- ***, *JHELP User Guide*: <http://java.sun.com/products/javahelp/>
- ***, *Internationalization Tutorial*:
<http://java.sun.com/docs/books/tutorial/i18n/index.html>
- ***, *Specificația Unicode*: <http://www.unicode.org>
- ***, *Tomcat*: <http://jakarta.apache.org/tomcat>

Mai multe amănunte sunt disponibile pe situl dedicat acestei cărți la
<http://thor.info.uaic.ro/~stanasa/java/>.

La Editura POLIROM au apărut:

- Mircea Băduț – *Calculatorul în trei timpi*
- Emanuela Cerchez, Marinel Șerban – *PC. Pas cu pas*
- Silvia Curteanu – *PC – Ghid de utilizare*
- Luminița Fînaru, Ioan Brava – *Visual Basic. Primii pași... și următorii*
- Doina Hriniciuc Logofătu – *C++ . Probleme rezolvate și algoritmi*
- Marin Fotache – *SQL. Dialecte DB2, Oracle și Visual FoxPro*
- Sabin Buraga, Gabriel Ciobanu – *Atelier de programare în rețele de calculatoare*
- Emanuela Cerchez – *Internet. Utilizarea rețelei Internet. Proiectarea paginilor Web (Manual opțional pentru liceu)*
- Marin Fotache, Ioan Brava, Cătălin Strîmbei, Liviu Creșu – *Visual FoxPro. Ghidul dezvoltării aplicațiilor profesionale*
- Sabin Buraga, Victor Tarhon-Onu, Ștefan Tanasă – *Programare Web în bash și Perl*
- Dragoș Acostăchioaie – *Administrarea și configurarea sistemelor Linux*
- Mihail Voicu – *Introducere în automată*
- Doina Fotache – *Groupware. Metode, tehnici și tehnologii pentru grupuri de lucru*
- Dumitru Oprea, Dinu Airinei, Marin Fotache – *Sisteme informaționale pentru afaceri*
- Dragoș Acostăchioaie – *C și C++ pentru Linux*
- Sabin Buraga – *Proiectarea siturilor Web. Design și funcționalitate*
- Călin Ioan Acu – *Optimizarea paginilor Web*
- Ștefan Tanasă, Cristian Olaru, Ștefan Andrei – *JAVA de la 0 la expert*

în pregătire:

- Dinu Airinei – *Depozite de date*
- Mihai Cioată – *ActiveX. Concepte și aplicații*

www.polirom.ro

Coperta : Ionuț Broștianu
Tehnoredactor : Lucian Pavel

Bun de tipar : martie 2003. Apărut : 2003
Editura Polirom, B-dul Copou nr. 4 • P.O. Box 266, 6600, Iași
Tel. & Fax : (0232) 21.41.00; (0232) 21.41.11;
(0232) 21.74.40 (difuzare); E-mail: office@polirom.ro
B-dul I.C. Brătianu nr. 6, et. 7, ap. 33 • O.P. 37, P.O. Box 1-728,
70700, București; Tel. : (021) 313.89.78, E-mail: polirom@dnt.ro

Tiparul executat la S.C. LUMINA TIPO s.r.l.
str. Luigi Galvani nr. 20 bis, sect. 2, București
Tel./Fax : 211.32.60, 212.29.27, E-mail: lumina-lex@fx.ro
